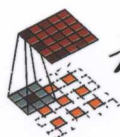


## 版权注意事项：

- 1、书籍版权归作者和出版社所有
- 2、本PDF仅限用于个人获取知识，进行私底下的知识交流
- 3、PDF获得者不得在互联网上以任何目的进行传播
- 4、如觉得书籍内容很赞，请购买正版实体书，支持作者
- 5、请于下载PDF后24小时内删除本PDF。

# 深度学习 轻松学



核心算法与视觉实践

冯超 (著)



中国工信出版集团



电子工业出版社  
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY  
<http://www.phei.com.cn>



# 深度学习 轻松学

核心算法与视觉实践

冯超 (著)

电子工业出版社

Publishing House of Electronics Industry

北京·BEIJING

## 内 容 简 介

本书介绍了深度学习基本算法和视觉领域的应用实例。书中以轻松直白的语言,生动详细地介绍了深层模型相关的基础知识,并深入剖析了算法的原理与本质。同时,书中还配有大量案例与源码,帮助读者切实体会深度学习的核心思想和精妙之处。除此之外,书中还介绍了深度学习在视觉领域的应用,从原理层面揭示其思路思想,帮助读者在此领域中夯实技术基础。

本书十分适合对深度学习感兴趣,希望对深层模型有较深入了解的读者阅读。

未经许可,不得以任何方式复制或抄袭本书之部分或全部内容。

版权所有,侵权必究。

## 图书在版编目(CIP)数据

深度学习轻松学:核心算法与视觉实践/冯超著.—北京:电子工业出版社,2017.7  
ISBN 978-7-121-31713-2

I. ①深…II. ①冯…III. ①人工智能①机器学习 IV. ①TP18

中国版本图书馆 CIP 数据核字(2017)第 121113 号

策划编辑:郑柳洁

责任编辑:郑柳洁

印 刷:北京季蜂印刷有限公司

装 订:北京季蜂印刷有限公司

出版发行:电子工业出版社

北京市海淀区万寿路 173 信箱 邮编:100036

开 本:720×1000 1/16 印张:21.75 字数:520 千字

版 次:2017 年 7 月第 1 版

印 次:2017 年 7 月第 1 次印刷

定 价:79.00 元

凡所购买电子工业出版社图书有缺损问题,请向购买书店调换。若书店售缺,请与本社发行部联系,联系及邮购电话:(010) 88254888, 88258888。

质量投诉请发邮件至 [zltts@phei.com.cn](mailto:zltts@phei.com.cn), 盗版侵权举报请发邮件至 [dbqq@phei.com.cn](mailto:dbqq@phei.com.cn)。

本书咨询联系方式:(010) 51260888-819 [faq@phei.com.cn](mailto:faq@phei.com.cn)。

# 前言

从你拿起这本书的那一刻开始，我就和你一起踏上了这段有关深度学习的冒险之旅。本书中有简单直白的叙述，也有复杂冗长的证明；有调皮的调侃，也有深刻的沉思。所有的一切，都是为了帮助你更轻松地对深度学习有更多了解、更多感悟。

本书的前身是我在知乎上的《无痛的机器学习》专栏（<https://zhuanlan.zhihu.com/hsmmy>）。作为在路上前进的一枚“小白”，我一直有写读书笔记的习惯。早期，我会把笔记写在阅读的文献上，在这些篇章的关键处做标记、写好注释。时间一长，当我想再读这些曾经阅读过的文献时，那些注释竟然变得莫名其妙起来。当初难以理解的概念又回到难以理解的状态，于是我不得不再次阅读原文，重新获得其中的感悟，这样周而复始，一次次在这个循环中打转。

后来，我选择彻底甩掉曾经阅读过的文献——既然它给我带来了那么多痛苦，为什么每次回顾时还要与它相对？于是我开始尝试将文献的所有内容重新组织成一篇短小的文章，文章中列出文献中的关键内容。这种文章简明扼要、直击重点，看着很清爽，可以帮助我快速回顾阅读的内容，又不用再回到原文，所以这种方法很快就替代了以前的那种方法。

再后来，我发现了新的问题。虽然我摆脱了晦涩的论文内容，但摆脱不了自己重述的文字。尤其是这些浓缩后的“精华”文字，理解起来并不比原始文献强多少，而且由于缺少很多细节，一旦遇到理解困难的状况，我不得不回到原文搜寻答案，于是这样的痛苦又得经历一次。

这时，我想起了父亲曾对我的教导：“书要先读厚，然后再读薄”。所谓的读厚，就是给阅读的书增加更多的内容，例如注释、自己的理解等。当自己能完全掌握书中的内容时，再将书中的细枝末节慢慢砍掉，将主体思想保留在心中，这就是读薄的过程。这样就完成了对一本书的深入阅读，其中读薄的过程和《倚天屠龙记》中张三丰教授张无忌太极拳剑的过程很相似。如果站在信息论的角度看，所谓的“重意不重形”，可以看作保证低频信息的正确性，依概率分布生成高频信息的一种“招式”生成模式。能达到这等境界，方可谓之大成。

对我来说，面对潮水般涌来的深度学习知识，想淡定从容应对是不可能的。也就是说，一开始就把书读薄是不可能的。所以，浓缩版的总结文章也慢慢被我否定了。那么

只剩下一个选择了，就是把书读厚，把文章写得拖沓冗长一些。于是我开始尝试用拖沓但细致的语言做文献的总结，经过一次次地尝试，文章的易读性确实有了明显提升，因为文章描述不清楚而回看原文的次数也越来越少。与此同时，我发现写作这个过程让我对问题有了更深入的理解。

拖沓式写法通过写作驱动，更容易发现知识点与知识点之间的沟壑。要想详细描述自己阅读的内容，就得确定文章中的逻辑自己是否真的明白了。有时探究这些逻辑所花费的时间远比自己想象得长，但是它确实让我感受到了进步与成长。

渐渐地，拖沓式文章越写越多，我也逐渐将其中一些文章公开，希望能够与大家分享：希望大家能和我对同一个问题产生共鸣，同时也可以在阅读中指出文章中的错误。不到一年，我见证了我写的文章被越来越多的人关注、讨论，在交流的过程中我收获了又一次成长。

在完成了几十篇文章后，本书的编辑郑柳洁女士联系我，问我是否有兴趣将这些文章集结，变成一本出版印刷的书。在此之前我并没有仔细想过这件事儿，但是她的建议让我觉得这也许是又一次证明自己、使自己成长的机会，而且出版书籍在我心中也是一件神圣的事儿，于是我接受了这个建议，开始了整理书稿的工作。

整理书稿并没有想象中那么简单。网上的文章都是单独出现的，而书中的文章需要有一定的整体性；网上的文章可以用比较随意的语言叙述，而书中的语言需要尽量正式、客观。这些挑战使我修改了放在网络上的很多文字，为了确保表达清晰准确、语言通顺流畅，有些文章基本被重新写了一遍。整理这些文章花费了很多业余时间，但功夫不负有心人，这项工作被一点点地完成了。

本书主要介绍了深度学习，尤其是卷积神经网络相关的基础知识，以及计算机视觉的部分应用内容。书中既包含深度学习中的基础知识，也包含部分进阶内容，同时也包含一些较新颖的概念与模型，适合不同人群阅读。对初学者来说，本书十分详细地介绍了很多基本概念，对自学入门深度学习很有帮助；对有一定经验的从业人员来说，本书可以梳理领域内的知识点，也可以作为工具书使用。

本书中的示例代码由 C++ 和 Python 两种语言实现，读者只要对两种语言有基本的了解即可。本书主要使用了 Caffe 这个第三方开源框架，在此向 Caffe 的作者表示感谢。现在有很多优秀的开源框架，这些框架各有优劣，但本质上有很多共性。对没有使用过 Caffe 的读者来说，阅读本书时不会有大的困扰。

最后聊聊本书的书名。最初想使用“无痛”这个词，无奈这个词太容易引发联想，不适合作为一本严肃的出版物的书名。“学习”这个词自古以来就不是轻松的代名词。“学习”的“习”字曾写作“習”，意思为鸟类挥动翅膀一次次试飞，其中暗含了反复练习的过程；在日语中“学习”被写作“勉強する”，从大家能看懂的两个字就可以看出，学

习中的艰辛；在韩语中“学习”又被写作“공부하다”，它前两个字的发音和“恐怖”二字很像，也许当初这个词背后的含义和恐怖有关。这样看来，东北亚的这几个国家都学习这件事都充满了敬畏之心，学习这件事是绝对不会轻松的，更不会是无痛的，经历过多年教育的读者相信也会有体会。

学习的过程充满痛苦，而这是不可避免的。我们每个人在人生的道路上都有过不断探索、不断遇到挫折，然后改进自我，完成进化的体验。就像机器学习中被训练的模型一样，不断地完成预测、找到差距、更新参数，最终达到最优。在这条自我成长的道路上，每一次的失败、每一次的努力都充满了艰辛，然而没有痛苦的积累，就不会有最终快乐的进发。

人生苦短。虽然人生要经历许多的痛苦，但我们的目标并不是痛苦，而是走向彼岸的快乐。因此，如果能有减少痛苦的良方，大家还是愿意一试。社会的不断发展进步就是探寻减少痛苦的解药，而本书的写作目标也是希望本书能成为各位读者学习路上的解药，在减少痛苦的同时实现心中的目标。

感谢邓澍军、夏龙先生在本书内容创作期间对内容的严格审核并提出宝贵的意见和建议；感谢知乎上各位为《无痛的机器学习》专栏中系列文章指出错误、提出疑问的朋友，是你们让文章变得更严谨；由于本人才疏学浅，行文间难免有所纰漏，望各位读者多多包涵，不吝赐教。最后感谢所有关心、支持我完成这件不易的工作的亲人和朋友。我爱你们！

作者

# 阅读须知

## 本书图片

部分图片可能需要放大观察，纸面上无法呈现应有效果。为此，书中图片均在博文视点官方网站提供下载。

## 读者服务

轻松注册成为博文视点社区用户（[www.broadview.com.cn](http://www.broadview.com.cn)），您即可享受以下服务。

- **下载资源：**本书所提供的示例代码及资源文件均可在 [下载资源](#) 处下载。
- **提交勘误：**您对书中内容的修改意见可在 [提交勘误](#) 处提交，若被采纳，将获赠博文视点社区积分（在您购买电子书时，积分可用来抵扣相应金额）。
- **与作者交流：**在页面下方 [读者评论](#) 处留下您的疑问或观点，与作者和其他读者一同学习交流。

页面入口：<http://www.broadview.com.cn/31713>



# 目录

<b>1 机器学习与深度学习的概念</b>	<b>1</b>
1.1 什么是机器学习	1
1.1.1 机器学习的形式	2
1.1.2 机器学习的几个组成部分	8
1.2 深度学习的逆袭	9
1.3 深层模型在视觉领域的应用	13
1.4 本书的主要内容	15
1.5 总结	17
<b>2 数学与机器学习基础</b>	<b>18</b>
2.1 线性代数基础	18
2.2 对称矩阵的性质	22
2.2.1 特征值与特征向量	22
2.2.2 对称矩阵的特征值和特征向量	23
2.2.3 对称矩阵的对角化	24
2.3 概率论	25
2.3.1 概率与分布	25
2.3.2 最大似然估计	28
2.4 信息论基础	31
2.5 KL 散度	33
2.6 凸函数及其性质	37
2.7 机器学习基本概念	39
2.8 机器学习的目标函数	42
2.9 总结	44

<b>3 CNN 的基石：全连接层</b>	<b>45</b>
3.1 线性部分	45
3.2 非线性部分	48
3.3 神经网络的模样	50
3.4 反向传播法	55
3.4.1 反向传播法的计算方法	55
3.4.2 反向传播法在计算上的抽象	58
3.4.3 反向传播法在批量数据上的推广	59
3.4.4 具体的例子	63
3.5 参数初始化	65
3.6 总结	68
<b>4 CNN 的基石：卷积层</b>	<b>69</b>
4.1 卷积操作	69
4.1.1 卷积是什么	69
4.1.2 卷积层效果展示	73
4.1.3 卷积层汇总了什么	76
4.1.4 卷积的另一种解释	77
4.2 卷积层的反向传播	79
4.2.1 实力派解法	80
4.2.2 “偶像派”解法	84
4.3 ReLU	88
4.3.1 梯度消失问题	89
4.3.2 ReLU 的理论支撑	92
4.3.3 ReLU 的线性性质	93
4.3.4 ReLU 的不足	93
4.4 总结	94
4.5 参考文献	94



<b>5 Caffe 入门</b>	<b>95</b>
5.1 使用 Caffe 进行深度学习训练	96
5.1.1 数据预处理	96
5.1.2 网络结构与模型训练的配置	100
5.1.3 训练与再训练	108
5.1.4 训练日志分析	110
5.1.5 预测检验与分析	112
5.1.6 性能测试	115
5.2 模型配置文件介绍	117
5.3 Caffe 的整体结构	122
5.3.1 SyncedMemory	124
5.3.2 Blob	125
5.3.3 Layer	125
5.3.4 Net	126
5.3.5 Solver	126
5.3.6 多 GPU 训练	127
5.3.7 IO	127
5.4 Caffe 的 Layer	128
5.4.1 Layer 的创建——LayerRegistry	128
5.4.2 Layer 的初始化	130
5.4.3 Layer 的前向计算	132
5.5 Caffe 的 Net 组装流程	133
5.6 Caffe 的 Solver 计算流程	139
5.6.1 优化流程	140
5.6.2 多卡优化算法	142
5.7 Caffe 的 Data Layer	145
5.7.1 Datum 结构	145
5.7.2 DataReader Thread	147
5.7.3 BasePrefetchingDataLayer Thread	148

5.7.4	Data Layer . . . . .	149
5.8	Caffe 的 Data Transformer . . . . .	150
5.8.1	C++ 中的 Data Transformer . . . . .	150
5.8.2	Python 中的 Data Transformer . . . . .	153
5.9	模型层扩展实践——Center Loss Layer . . . . .	156
5.9.1	Center Loss 的原理 . . . . .	156
5.9.2	Center Loss 实现 . . . . .	160
5.9.3	实验分析与总结 . . . . .	164
5.10	总结 . . . . .	165
5.11	参考文献 . . . . .	165
6	深层网络的数值问题 . . . . .	166
6.1	ReLU 和参数初始化 . . . . .	166
6.1.1	第一个 ReLU 数值实验 . . . . .	167
6.1.2	第二个 ReLU 数值实验 . . . . .	169
6.1.3	第三个实验——Sigmoid . . . . .	171
6.2	Xavier 初始化 . . . . .	172
6.3	MSRA 初始化 . . . . .	178
6.3.1	前向推导 . . . . .	178
6.3.2	后向推导 . . . . .	181
6.4	ZCA . . . . .	182
6.5	与数值溢出的战斗 . . . . .	186
6.5.1	Softmax Layer . . . . .	186
6.5.2	Sigmoid Cross Entropy Loss . . . . .	189
6.6	总结 . . . . .	192
6.7	参考文献 . . . . .	192

<b>7 网络结构</b>	<b>193</b>
7.1 关于网络结构，我们更关心什么	193
7.2 网络结构的演化	195
7.2.1 VGG：模型哲学	195
7.2.2 GoogLeNet：丰富模型层的内部结构	196
7.2.3 ResNet：从乘法模型到加法模型	197
7.2.4 全连接层的没落	198
7.3 Batch Normalization	199
7.3.1 Normalization	199
7.3.2 使用 BN 层的实验	200
7.3.3 BN 的实现	201
7.4 对 Dropout 的思考	204
7.5 从迁移学习的角度观察网络功能	206
7.6 ResNet 的深入分析	210
7.6.1 DSN 解决梯度消失问题	211
7.6.2 ResNet 网络的展开结构	212
7.6.3 FractalNet	214
7.6.4 DenseNet	215
7.7 总结	217
7.8 参考文献	217
<b>8 优化与训练</b>	<b>219</b>
8.1 梯度下降是一门手艺活儿	219
8.1.1 什么是梯度下降法	219
8.1.2 优雅的步长	220
8.2 路遥知马力：动量	225
8.3 SGD 的变种算法	232
8.3.1 非凸函数	232
8.3.2 经典算法的弯道表现	233
8.3.3 Adagrad	234

8.3.4	Rmsprop	235
8.3.5	AdaDelta	236
8.3.6	Adam	237
8.3.7	爬坡赛	240
8.3.8	总结	242
8.4	L1 正则的效果	243
8.4.1	MNIST 的 L1 正则实验	244
8.4.2	次梯度下降法	246
8.5	寻找模型的弱点	251
8.5.1	泛化性实验	252
8.5.2	精确性实验	255
8.6	模型优化路径的可视化	255
8.7	模型的过拟合	260
8.7.1	过拟合方案	261
8.7.2	SGD 与过拟合	263
8.7.3	对于深层模型泛化的猜想	264
8.8	总结	265
8.9	参考文献	265
9	应用：图像的语言分割	267
9.1	FCN	268
9.2	CRF 通俗非严谨的入门	272
9.2.1	有向图与无向图模型	272
9.2.2	Log-Linear Model	278
9.2.3	条件随机场	280
9.3	Dense CRF	281
9.3.1	Dense CRF 是如何被演化出来的	281
9.3.2	Dense CRF 的公式形式	284
9.4	Mean Field 对 Dense CRF 模型的化简	285
9.5	Dense CRF 的推断计算公式	288

9.5.1	Variational Inference 推导	289
9.5.2	进一步化简	291
9.6	完整的模型: CRF as RNN	292
9.7	总结	294
9.8	参考文献	294
<b>10</b>	<b>应用: 图像生成</b>	<b>295</b>
10.1	VAE	295
10.1.1	生成式模型	295
10.1.2	Variational Lower bound	296
10.1.3	Reparameterization Trick	298
10.1.4	Encoder 和 Decoder 的计算公式	299
10.1.5	实现	300
10.1.6	MNIST 生成模型可视化	301
10.2	GAN	303
10.2.1	GAN 的概念	303
10.2.2	GAN 的训练分析	305
10.2.3	GAN 实战	309
10.3	Info-GAN	314
10.3.1	互信息	315
10.3.2	InfoGAN 模型	317
10.4	Wasserstein GAN	320
10.4.1	分布的重叠度	321
10.4.2	两种目标函数存在的问题	323
10.4.3	Wasserstein 距离	325
10.4.4	Wasserstein 距离的优势	329
10.4.5	Wasserstein GAN 的实现	331
10.5	总结	333
10.6	参考文献	334



# 1

## 机器学习与深度学习的概念

### 1.1 什么是机器学习

当你拿起本书时，我们已经携手踏上了探索机器学习的征程。机器学习作为人工智能的一个分支，近年来在科研和工业界有了很大的发展，同时也引起了圈内和圈外所有人的关注。现在，机器人的一些技能已经完全超出了旁观群众的认知，这使得全民又一次开始讨论“机器是否能够取代人类”这样的话题。

“机器取代人类”虽然是一个听上去让人不太舒服的话题，但是这个事情本身是令人向往的。最近几年，机器人在眼睛、耳朵、嘴巴和四肢方面的技术都有了很大提升，在某些领域，它的能力已经超过了一些专家的水平；在另一些领域，其能力也逼近专家的水平；在某些领域，其能力已经有了令人惊叹的重大突破。这些提升让人感到欣喜，因为机器人的表现离大家想象中的人工智能又近了一步。不过，从更深层的角度分析近几年机器学习的发展，我们又会发现一个残酷的事实：这几年深度学习的发展并不是完全得益于机器学习理论研究的新突破，主要是因为支撑理论的外部条件有了很大的提升。

让我们先回到机器学习本身，回到那些经过历史积淀的理论本身，看看它们究竟给我们设定了一个怎样的世界观，它们又有哪些启发我们的思维模式。其实，机器学习理论数十年没有本质变化这件事儿，从某种意义上说也是件好事，说明这个思维框架经得住时间的考验，也说明了它被学习的价值。

1.1.1 机器学习的形式

大家都知道，世界上的许多事物都可以从很多角度、用很多方法进行描述：我们可以十分详细地描述一个事物的细节，以图 1-1 所示的一朵花为例，我们可以说出这朵花的形状、颜色，花瓣有几朵，什么时节开花；也可以概括地描述一个事物，把一个抽象的名字冠在这种花上，以后见到这类花就统一叫它们这个名字。



图 1-1 花

同样，我们可以描述一间房子的地理位置、周围环境、房屋使用面积和配套设施等，如图 1-2 所示；也可以用一个价格标记这个房屋的租赁价格。除了这些描述，我们还可以从时间和空间等不同角度对同一个事物做出不同的描述。



图 1-2 房子的外观图

很显然，这些描述都是针对同一个事物，那么描述之间必然存在某种关联关系。人类经过长时间的学习事件，已经可以在自己熟悉的领域轻松地完成这些描述的转换，而对于计算机，想让它在短时间内完成这些转换几乎是不可能的。它要通过人类的操作和指引才能具备这样的能力。在现实中，我们通常会遇到这样的问题：对于一个事物，关于它的最直观的描述可以以一种比较容易的方式获得，我们希望利用描述间的相关性，将这种描述转换成另一种相对抽象且不容易直接获得的描述。更进一步地，我们希望这个分析运算的事情能够由计算机自动完成，因为计算机运算速度快，每秒可以进行大量的数值计算；同时表现稳定，不知疲倦，没有情绪波动，给电就能工作，成本比人低，所以一旦将计算机的计算潜力激发出来，它就可以更好、更快地完成人类可以解决或者人类不易解决的问题。

因为现代的计算机主要基于数字计算实现，所以我们需要把要解决的问题全部以数字的形式表达出来。数值计算是计算机的强项，但是想把所有信息都用数字的形式表达出来还是有点难度的。为此，前辈们想尽了各种办法。比方说，想表示一朵花的大小，可以把这朵花想象成一个圆形，花蕊是这个圆形的圆心，然后测量这个圆的半径，那么这个半径就是一个可以被计算机使用的描述信息，这个信息在机器学习中一般被称作特征（Feature）。

大小这个属性比较好转化为特征，而且这个转化后的特征和我们的直觉吻合。然而，实际中有些事物就不那么好表示了。比方说，我们想把不同的中文词语用特征的方式表示出来，这件事情就会遇到不小的麻烦。由于特征最终需要以数值的方式表达，而数值天生具有可比较性，也具有度量这样的概念，那么词语与词语之间的度量该如何表示呢？如果像花朵大小那样，把所有的汉字映射到正整数数轴上，每一个汉字确实有了自己的数值表示方法，但是词语之间的比较就失去了意义。或者说，采用传统的定义在欧式距离下的距离测量方法变得不再适用。试想一下，如果我们有一个词语叫“如果”，它被映射到 200，接着又有一个词语叫“但是”，它被映射到 300，那么它们在欧式空间的距离：

$$\sqrt{(300 - 200)^2} = 100$$

表示什么意思呢？如果刚好有一个词语被映射到了 100，那么这个词语是不是就是“如果”和“但是”的距离？还是说，这个结果有其他含义呢？这些问题一下子变得难以回答，至少和花朵大小这件事儿相比，这个距离难以解释。

那么，中文词语有什么可以表示的方法呢？一种比较简单的表示法被人们称为 **One Hot 编码**（One-Hot encoding），它的表示方法如下。

1. 可以假设中文词语的个数是有限的，或者说我们考虑的中文词语集合是全体中文词语的一个子集，这个子集是有限的，这里假设一共有  $N$  个词，于是我们定义一个维度为  $N$  的空间，这个空间的每一维只能取两个值：0 和 1。



2. 空间中的每一维代表一个词语，于是对于每一个词语，它都可以找到自己在这个空间的位置，也就是说，每一个词语都可以找出一个向量，这个向量在所有词语中排第  $m$  个，那么这个向量的第  $m$  位为 1，其余位为 0。这样，每一个词语都可以对应这个高维空间中的一个点，这个点的坐标就用上述方式构建的向量表示，如图 1-3 所示。

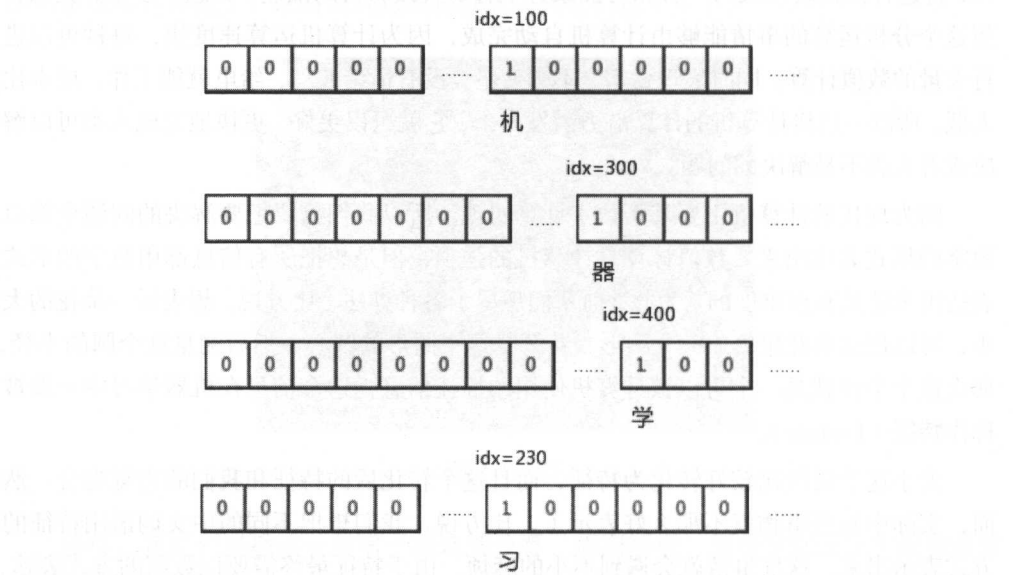


图 1-3 One Hot 编码形式

空间构建好了，下面该定义度量了。采用了这个空间后，度量的定义变得简单了许多。我们惊奇地发现，借用欧式空间中的距离计算方法衡量每一对词语之间的距离，每一对不同词语之间的距离是相同的，且大于 0。从最直观的角度来看，由于每一对词所使用的文字不同，因此它们存在距离，同时用一个统一的距离表示词语间的差异也是可以理解的。但是从语意的角度来看，这样的距离度量显然是不合理的。基于人类的常识系统，不同的词语对之间的距离有可能是不同的。有些词语对之间的关系十分密切，有些词语对之间的关系则显得十分疏远，这样的关系并没有被这个空间表达出来，不过这个空间还是要比之前的正整数数轴强大不少，起码它的距离可以解释得很直观。

从上面的例子可以看出，中文词语存在于某个难以描述的空间中，这个空间里有一套自己的运转规则和体系，它的规则和体系与我们常见的空间不太相同。为了让计算机能够进行中文词语相关的工作，我们需要构建一个空间来表示这些词语。在构建空间的过程中有两个基本要求：一是要用当今计算机体系能够接受的形式表示；二是尽可能地和原始的空间靠近。刚才的 One Hot 编码方法满足了第一条，但是第二条做得

并不好。这就好像我们在计算机上重新描绘了这个空间，但是丢失了一些原始空间的特性。想要更好地复原原始空间的内容，实际上还有很长的路要走。

前面介绍了许多关于特征描述量化的问题，下面就要看看描述之间应该被如何转换。前面提到同一个事物可以从不同角度以不同的细致程度被描述。一个事物可以被很简单地描述，也可以被很详细地描述；一个事物可以从一个角度描述，也可以从另一个角度描述。有时，我们需要从一个角度的描述转换到另一个角度，例如收到了关于一朵花的描述，读者可以从花的描述中知道被描述的花的种类。这是一种从具体描述到泛化描述的过程。同时，也可以从一个泛化的描述找寻具体的描述，例如从一朵花的花名联想到这种花的模样。

很显然，我们之所以可以完成两种描述的转换，是因为两者本质上描述了同一个事物，它们存在着很强的相关关系。在现实生活中，找到这样的关系并完成两者之间的转换是十分有必要的。人类每天都在做这种模式的事情，其中一种描述相对容易获得，而另一种描述可以通过转换获得，人类也希望机器能够帮助他们完成这样的工作。实际上，这个形式和我们曾经学习过的函数十分相像。函数的功能就是把输入经过某种变换转换成输出，在这个场景下，容易获得的特征一般作为函数的输入，期望得到的特征一般作为函数的输出。拥有了输入和输出，剩下的工作就是确定这个中间的映射。可以说，绝大部分的机器学习都遵循了这个模式，在机器学习中，一般将这个“映射”称为模型。

从总体上看，模型充当了映射的作用，但是映射的表现形式也有很多种。机器学习常见的学习方式有三种：监督学习、非监督学习和增强学习。监督学习和非监督学习的形式相对简单，我们先介绍这两种。在监督学习中，部分模型输入和对应的理想输出是已知的，这些输入输出通常是整个问题中所有输入输出的一个代表。由于理想输出已经知道，所以我们可以比较它和模型的输出结果判断模型的表现和期望是否一致，如果一致，那么证明模型表现很好；如果不一致，也可以从中发现模型输出和正确答案的差距，这样模型就可以在输入输出的“监督”下不断学习，让自己的结果更靠近“标准答案”。在非监督学习中，通常标准输出是未知的，我们不能判断模型的输出是否“完全正确”，只能通过其他的方法辅助判断结果的正确程度，所以它被称为非监督学习。监督学习和非监督学习的一般形式如图 1-4 所示。

增强学习与前面两种方法不同，在这种学习方法中，模型需要对依时间排列的一连串输入状态（State）做出响应。当模型对指定的输入返回结果后，一个外部的环境（Environment）会对模型返回结果做出响应，并返回两个结果。

1. 对模型的结果做评价，返回一个奖励（Reward）作为模型表现的评价。这个评价同监督学习的目标函数结果不同，它并不直接衡量模型结果与标准答案的差距，

而是以一种奖励的形式返回，如果表现好那么奖励就会大，反之则会小。通常，这个奖励不但和当前模型的表现有关，还和模型在过去时间里的表现有关。

- 2. 与此同时，环境还会返回一个新的输入（State），模型也会对这个输入开始新一轮的响应。

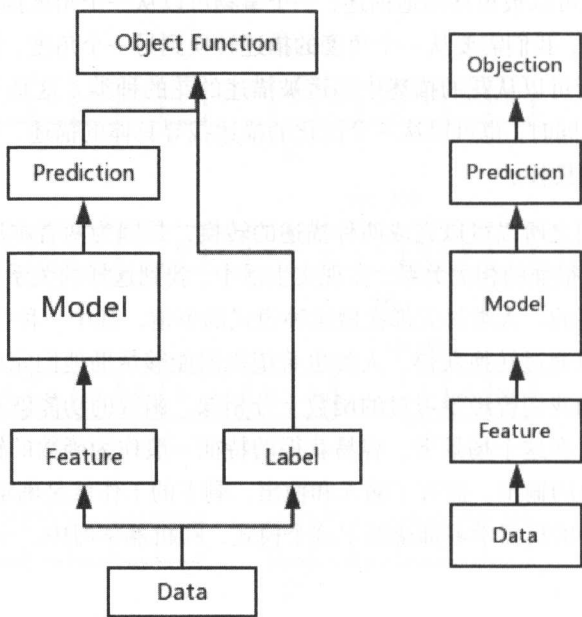


图 1-4 监督学习与非监督学习的流程

增强学习的模型架构比较复杂，这里就不再详细介绍了，上面介绍的内容如图 1-5 所示。

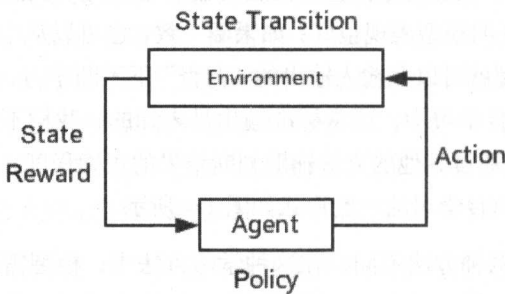


图 1-5 增强学习

这三种学习方法是凭空产生的吗？当然不是。实际上，这三种学习方法和人的学习方法高度吻合，可以说每个人在自己的一生中都可能经历过这三种学习方式。

**监督学习：**大家在学校课堂里学到的知识绝大多数是通过监督学习学到的。 $1+1$ 等于几？等于2。这样的问题都是有答案的，而答案就是关于问题的另外一种描述。在学校里，学生充当了函数的功能，如果学生给出的结果和标准答案不一样，学生就会被教育如何改正自己的错误，从而给出正确的答案。这个“做题 → 看答案 → 改错 → 再做题”的过程和机器学习中模型训练的过程一致，只不过在这里训练的对象是我们自己。

**非监督学习：**这个部分在学校课堂上遇到得比较少，反而是在研究机构比较多。这类问题通常对应着“某某猜想”，或者是通过实验和分析得出某个未知的结论。由于没有训练的过程，而且不知道一个问题的答案，我们只能利用现有知识和现有的经验解决问题，并用一些间接的方式评价模型给出的结果。看上去，这一部分的难度比前面的监督学习要困难不少，不过一旦我们解决了这些问题，发现了其中的规律，这些问题也就有了标准答案，我们也就可以按照监督学习的方式学习这些知识了。

**增强学习：**这种学习方式通常不会出现在课堂上，而是出现在课堂以外的地方，比方说人的言行和决定。同课堂上的场景不同，每个人的一举一动往往不会有标准答案，而且很多行为发生之后，周围的环境会根据这些行为产生一定的反馈。例如，某个人产生了一个行为后，同学对这个行为的态度，行为之后发生的事情等。这些反馈并不明确指出行为的对错，但是它依然改变了周围的环境，让行动者看到了结果的好与坏，从而揭示了行为正误的程度。有人曾说过“小孩才分对错，大人只看利弊”，这也从侧面说明监督学习和增强学习之间的差异。

有这三种学习方式，我们也就能见到擅长这三种学习方式的人群。实际上，在学校里我们就能感受到。

**学霸：**学霸是监督学习造就的模范产物。学霸最擅长的就是监督学习，说得再通俗一点，就是“刷题”。所谓的刷题就是不断地完成从输入到已知输出映射的学习过程。理论上，“有效率地”刷越多的题，学霸的学习成绩就越好，如何“有效率地”刷题，和如何“高效地”完成模型训练类似。所以像“功夫不负有心人”这样的谚语用在学霸身上十分合适。

**学神：**学神可以算是非监督学习的典型。他们总可以解决一些极为困难的问题，而他们的思维能力和分析能力往往很强。换成机器学习的术语来说，就是他们对各种输出未知问题的建模能力很强。

**班干部：**班干部或者社团干部可以算是增强学习的成功典范。这部分人群往往很懂人情世故、待人接物，在学校里显得更成熟，看着更像“社会人”“老江湖”。他们的这些能力得益于平时对周围环境的理解，因为更关心这些反馈，才使得他们在这方面有了很大的提高。

所以这么看起来，三种学习实际上都和我们密切相关。机器学习的思路 and 人类其实差不多。和人类更相近的是，每一种人都有自己擅长的能力和擅长的学习方法，机器模型也不例外，我们要针对不同的问题建立不同的模型，这样才能把问题解决得更好，也让模型发挥得更好。

### 1.1.2 机器学习的几个组成部分

虽然前面已经洋洋洒洒地聊了几千字，但是我们还是没有把机器学习的模式完全介绍清楚。不过没有关系，我们已经介绍了机器学习的一个关键问题：机器学习和人类的学习方式十分相近。这带来了一个好消息：我们只要按照人类的学习方法，把所有必要的部件准备好，机器就能够学习，而且由于机器不知疲倦，给电就可以工作，它可以无休止地学习下去，这样恐怖的学习干劲绝对会让人类感到害怕。当然，前面也提到了，机器学习的一大难点就是把所有问题用机器可以理解的形式表述出来，所以机器替代人类的时间来得不会太快。

那么，人类学习需要哪些部件呢？我们就以学校里最常见的监督学习为例解释。

- **人。**学生就是要训练的目标，可以将他们看作一群智能体，具有自主分析判断问题的能力。对于机器学习来说，我们需要人为地构建起一个像人类大脑一样的“智能体”，才能让它像人一样解决问题。除此之外，人能够接收各种各样的信号，也需要让机器接收类似的输入描述才行。
- **试题。**对于学生来说，这就是要完成的题目；对机器来说，这就是要输入的描述特征。
- **答案。**对于学生来说，这就是大家常见的正确答案；对机器来说，这就是理想中应该给出的输出。
- **评分标准。**对学生来说，评分标准可以帮助我们衡量一个学生的能力，也能准确定位他们的不足之处；对于机器来说，需要定义一个个“目标函数”告诉机器它给出的答案和标准答案之间的差距，从而帮助机器定位自己的不足。
- **改进方案。**对学生来说，当发现了他们的不足之处，我们需要分析并告诉他们哪里有问题，哪里要提高，从而修正他们已经学到的认知和知识，从而使他们表现得更出色。当然，如果改进方案不够出色，学生的成绩可能会退步，这也是常见且难以避免的。对机器来说，我们需要针对模型的结构使用与之匹配的优化算法来确保模型根据损失函数的情况更新函数内部信息，从而达到提升模型能力的目标。

总的来说，我们可以把上面的五点归纳为监督学习场景下机器需要的四个要素：模型、数据、目标函数（损失函数）和优化算法。

对研究机器学习理论的人来说，大家主要关注的是其中的模型、目标函数和优化算法。实际上，众多机器学习书籍主要围绕着这三个问题讲述。它们讲述每一个模型适用的范围，讲述目标函数和模型的搭配关系，讲述不同优化算法的特点和效果。本书也将围绕这三个方面介绍很多有意思、有趣的话题，本书主要关注监督学习问题。

## 1.2 深度学习的逆袭

1.1 节介绍了机器学习的几个组成部分，本节将围绕这些组成部分讨论另一个很重要的问题：为什么深度学习能够在这些年大放异彩，支撑它取得巨大成功的究竟是其中的哪些部分呢？

在机器学习刚刚兴起时，大家普遍认为机器学习模型只能解决一些相对简单的问题。由于数据采集比较困难，针对这些问题的数据也比较少，数据的缺失对研究机器学习十分不利。好在问题本身比较简单，只需要对问题中的数据做简单的处理就可以了，解决问题所需要的模型也相对简单。

再后来，人们发现机器学习在解决简单问题上的表现不错，于是开始尝试一些更复杂的问题。可惜的是，由于条件限制，与这些更复杂问题相关的数据依然有限，这时训练数据成为了瓶颈。为了更好地利用有限的的数据解决问题，前辈们花了大量的心血设计各种复杂精巧的模型，那个时代涌现出了很多理论完备且效果突出的模型，比方说大名鼎鼎的支持向量机（Support Vector Machine）。

随着时间的不断发展，大数据时代来临，我们面对的很多问题终于有了充足甚至过量的数据支持。有了足量的数据支持，就可以从数据中更清晰地看出问题，那么这时模型的压力就变得小了很多，我们可以使用一些相对简单且扩展性足够好的模型，例如大规模特征的逻辑斯特回归（Logistic Regression）。这时，模型的深度和上一代模型类似，以浅层模型为主，于是对事物的描述特征就开始爆炸式的增长。特征工程在这个时代也成了机器学习工程师必备的技能之一。

那么，什么是特征工程呢？特征工程是通过一些操作将原始的特征转换成更容易被处理的特征。对于一些问题，如果原始特征不被处理，最终的效果可能会打折扣。举例来说，图 1-6 所示的这个汉字是什么字呢？

我们的输入是排列整齐的几百甚至几千个像素值的信息，如果直接使用这些像素值信息做监督学习，那么我们很有可能不会得到一个很好的结果。利用本书后面即将介绍的开源框架 Caffe 对手写数字数据集 MNIST 进行测试，模型采用一层全连接层和一个 Softmax 层，这样浅层的模型可以得到 0.89 的准确率。这样的精确率离工业上的应用还差许多。实验的模型如图 1-7 所示。

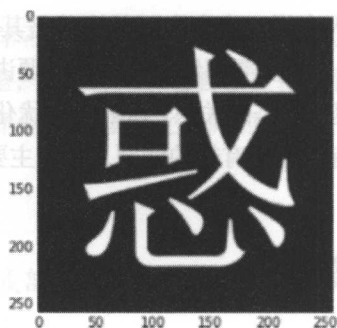


图 1-6 一个汉字的灰度图

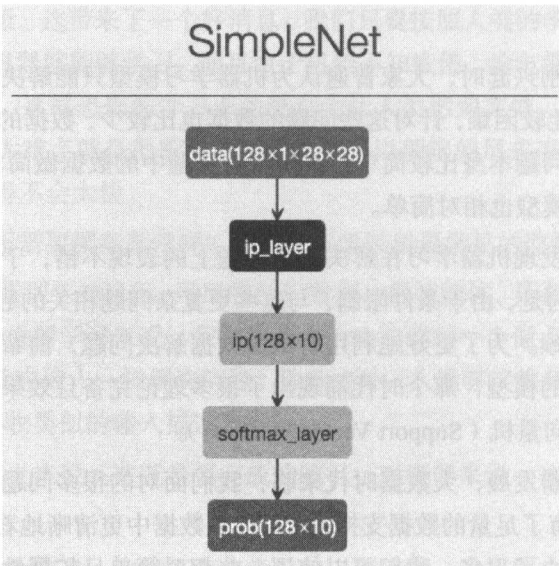


图 1-7 一个简单的浅层模型（由 <http://ethereon.github.io/netscope/#/editor> 生成）

这个实验告诉我们，想让计算机用比较浅层的模型在原始信息上解决问题是有些困难的，这不容易获得非常高的精度。如果把这些像素值排成一排放在那里，就算人类也不太容易识别出来。那么，在认出这些汉字的一瞬间，人类的大脑里都想了些什么呢？

科学研究发现，人类大脑的神经元会组成一个比较深层的网络结构，每一层的神经元肩负着不同的处理任务。举例来说，从视觉系统传来的信息要经过几个层次的处理才能得到正确的结果。可以想象每一层中，我们的大脑不同层级的神经元能处理不同复杂程度的信息，浅层的可以处理图像中局部的信息，深层的可以处理图像中大片区域的信息，信息随着神经元的不断处理而不断聚集，并不断变得易于分析和处理。



想要达到好的效果，模型也需要经过多个层次的处理才行。在深度学习爆发之前，科研人员主要使用浅层模型完成工作，于是前面几层神经元的工作由人类思考完成，人类负责利用算法把那些看上去抽象的信息变得更易于处理，最后人工设计处理得到的半成品再交给模型去学习。在图像识别和匹配问题中，我们有像 SIFT、HOG 等复杂的算法，它们既考虑到了多尺度（也就是不同分辨率下的）图像信息，又考虑了梯度这样的敏感信息，同时还有其他的信息聚合，这些工作相当于把人类大脑中前面几层神经元的工作完成了，这时用浅层模型就可以处理很多复杂的问题。所谓的特征工程也正是指这些特征生成的过程和方法，它让浅层模型可以更好地完成最后的工作。

为什么我们要自己设计这些算法，而不加大模型的深度，像人一样让它自己根据训练数据学习出来呢？这里有很多历史问题。深层模型曾经有两个难以攻克的问题，一个是计算量的问题。模型的深度越深，模型就变得更灵活，其中包含的参数也更多，想要训练这样的模型需要更大的计算量，这会比之前的浅层模型困难得多，因此训练这样的模型需要更大的计算量。即使当年我们的计算机已经十分先进，计算速度已经非常快，面对这样大的计算量还是有些力不从心。

另外一个问题是模型的复杂度成倍增长，导致模型在训练过程中变得不可控。浅层模型拥有的种种优良特性在这里不复存在。这里不妨把问题讲得更深入一些，从前面的介绍中已经知道，想要让模型拥有智能，需要使用适当的优化算法，让模型在训练时不断调整自身，提高能力。于是如何利用优化算法完成优化就显得十分重要。那么，之前的浅层模型有什么好处呢？浅层模型的损失函数往往可以满足凸函数的性质（后文会介绍），凸函数拥有非常好的特性，我们可以快速放心地优化函数而不用担心优化的过程中产生一些不可控的情况。比方说，人类会遇到越复习成绩越差的情况，而凸函数的优化一般不会遇到这样的问题。我们一旦采用了深层模型，凸函数的性质将不再满足，优化曲面也变得不那么友好，优化过程中许多问题就会冒出来。除此之外，模型训练中还有其他问题，这些问题会在本书的后面慢慢介绍。

摆在面前的这两座大山阻碍了深层模型的发展，以至于虽然深层模型很早就被人发明（其实从人脑的结构想象出这个结构并不困难），但是由于实现起来困难重重，理论方面又几乎是空白，于是这一分支一直没有被众前辈看中。现在时代不同了，机器的计算能力在不断增长，终于有一天，它的计算能力达到了可以优化深层模型的地步，同时模型的训练逐渐变得可控，应有的效果发挥了出来，于是深层模型终于重新出现在人们的视野中，大家开始重新审视这个被遗忘的屠龙神器。

一旦深层模型变得可以计算，研究人员就可以方便地做实验分析问题，这一领域的理论也就有了很大的发展，一些模型结构和优化方面的基本问题也逐渐被解开。虽然到目前为止，深层模型的理论依然不够完善，但是我们正在见证着一个个理论漏洞被补好，也许在不远的未来，深层模型的理论可以变得更完善。



前面说了这么多深层模型需要克服的问题，那么我们同样也要看看它的优点——也就是它受人热捧的原因。一个最直观的原因就是深层模型在效果上比上一代浅层模型厉害得多，在有些问题上可以说是碾压式的胜利。这对科研和工业界来说都是一个非常振奋的消息，不论黑猫还是白猫，能抓住老鼠就是好猫。有这么多高智商的专家，把弱点慢慢补上就行了，于是有些“旁门左道”的深层模型就这样火热起来了。

那么，深层模型究竟是如何碾压浅层模型的呢？说起这个事情，我们不得不提起2012年的那场比赛——ImageNet Large Scale Visual Recognition Challenge, 简称ILSVRC。在这一年的比赛中，由多伦多大学的 Alex Krizhevsky、Ilya Sutskever、Geoffrey Hinton 组成的 SuperVision 代表队在这一届的 Classification 和 Localization 竞赛中横空出世，强势碾压了其他对手，在最终的 Top5 Error 评价指标上比第二名的队伍低了 10% 以上，表 1-1 和表 1-2 是从比赛官网摘录的成绩。

表 1-1 分类任务的最终结果

队名	Top5 错误率	模型描述
SuperVision	0.15315	额外使用了来自 ImageNet Fall 2011 的数据进行训练
ISI	0.26172	以 SIFT+FV、LBP+FV、GIST+FV 和 CSIFT+FV 为特征的分类器加权融合而成

表 1-2 定位任务的最终结果

队名	Top5 错误率	模型描述
SuperVision	0.335463	额外使用了 ImageNet Fall 2011 的数据做分类训练
OXFORD_	0.500342	在从高层次 SVM 计算分数和 Baseline 分数混合选取的候选集上进行 DPM 重排序检测
VGG		

从上面的结果可以看出，在那一年，采用浅层模型的队伍使出了各种抽取特征的方法，可以说已经把浅层模型发挥到了极致，而第一名的模型只是作为深层模型的新手，就已经获得如此巨大的优势。如果有兴趣再看看 2016 年的数据，相信大家一定会感慨深度学习模型对浅层模型的碾压。

其实从原理上分析，我们也能解释深层模型强大的原因。深层模型的结构和人类大脑的层次结构更接近，从这个角度来看也确实具有更大的潜力，同时深层模型更能体现机器学习，尤其是统计机器学习的精髓——让数据说话。前面提到了浅层模型的套路，它们都需要一套复杂的特征工程使模型变得可用，这些复杂的特征工程是人为设计好的，相当于对原始信息做了一次筛选——哪些信息有用需要加工，哪些信息没

用需要丢弃。实际情况往往不是这样，丢弃的部分信息可能并不是完全没用，而留下加工的信息也不一定足够，人为设定的算法逻辑清晰易于实现，但总会留下一些盲区，这一部分也就成为了整个模型的短板。为了让这部分算法不成为短板，前辈们不断改进算法，但无论怎么改，算法的能力总是很难有质的飞跃。

深层模型就不同了，它并不需要设计算法，每一个层次产生的计算方式可能难以理解、难以描述，但它确实反映了当前数据的特点。它的模型参数看上去很不通用，但它确实反映了当前数据的特点。由于它可以很好地适配数据，所以只要它训练充分，模型将不再成为短板。

所以我们可以简单总结一下深度学习的优点：深度学习舍弃了特征工程的步骤，让模型更好地根据数据的原始状态学习成长，因此更容易学到数据中有价值的信息。只要我们能够解决好深层模型的学习问题，那么深层模型构建的重点将转移到数据身上——只要数据能够全面覆盖原始问题的范围，深层模型就能够从中学到原始问题的精髓。现在，数据已经不再是瓶颈，所以深层模型的实力超越浅层模型也就不奇怪了。

当然，我们前面提到的缺点依然笼罩在深层模型周围，这些难题成为了深度学习当下要攻克的目标，也成了我们需要学习掌握的内容。

### 1.3 深层模型在视觉领域的应用

说完了深层模型的优势，让我们简单了解深层模型都在哪些领域发挥了作用。由于本书主要关注视觉领域，下面就来看看深层模型在视觉方面的改变。

首先是**图像分类**。大家最熟悉的就是 ILSVRC 竞赛，竞赛中每个模型需要为一副图像提供 5 个可能的答案。只要这 5 个答案中有一个正确，就算模型识别正确。图 1-8 所示为比赛中的图片和模型猜测的结果。

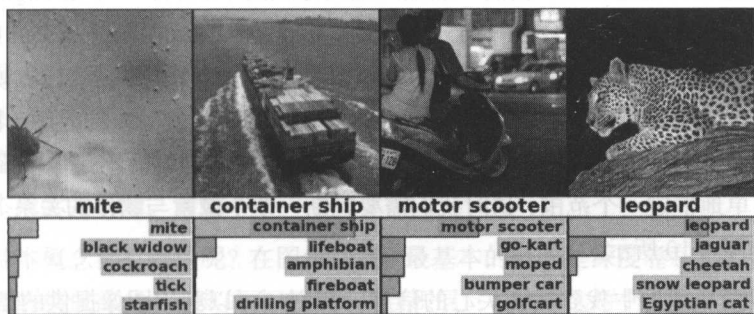


图 1-8 图像识别结果示例 [每一列展示了一张图片和对应的模型猜测的最可能的 5 个结果 ( 图片来源: Alex Krizhevsky, Ilya Sutskever, Geoffrey E. Hinton, ImageNet Classification with Deep Convolutional Neural Networks, NIPS, 2012. ) ]

从 2015 年开始，参与这个比赛的模型的识别率已经超越了人类，而且竞赛中涌现了一系列优秀的卷积神经网络模型，这些模型成为了科研界和工业界追捧的对象：从最早的 AlexNet，到后来的 VGG-Net、GoogLeNet，以及再后来的 Deep Residual Network，都成为了引领深度学习发展的典范。

介绍了图像分类后，我们再介绍与它很接近的目标检测问题。目标检测的例子如图 1-9 所示。

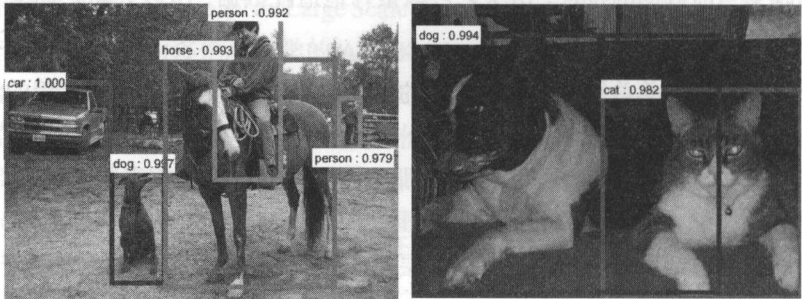


图 1-9 目标检测图像示例（图片来源：Shaoqing Ren, Kaiming He, Ross Girshick, Jian Sun, Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks, arXiv:1506.01497.）

在这样的日常图片中，机器模型要找出所有可能的实体，除了给出实体所在的范围，还要识别出实体的类别。这个问题算是一个十分困难的任务，除了本身千变万化的物体难以识别之外，物体与物体之间还存在着遮挡与相连的关系，这让定位和识别变得更困难。虽然困难，深层模型在这个领域还是获得了很大的成功。成功伴随着一批优秀的深层模型：从早期以滑动窗口为思想的 OverFeat 到基于图像分割再识别的 R-CNN（Regional-CNN）、Faster R-CNN，再到一次性搞定问题的 YOLO（You Only Look Once）和 SSD（Single Shot Detector），这些模型和其中蕴含的思想也给后人以启迪。

在图像处理和视觉领域，我们一般把问题划分成两个级别，一个是 High Level，另一个是 Low Level。这里的 High 与 Low 并没有高低之分，一般来说，High Level 的问题会从宏观的角度看待，而 Low Level 的问题则会从微观的角度看待。比方说，上面介绍的图像分类和物体定位，就是站在宏观的角度看待的；而接下来的应用（图像分割）则是从微观的角度分析的。我们需要精细地找到物体的边界，而不是像前面的物体定位那样，简单地框定一个范围，这时我们需要判断一个个像素与物体的关系。图像分割的示意图如图 1-10 所示。

除了根据从图像中找到我们关心的信息，我们还可以利用图像提供的信息生成不同形式的信息，比方说文字信息。这里又涌现出了一大批优秀的模型。如图 1-11 所示，我们可以根据图像中的信息生成描述图像的语言。

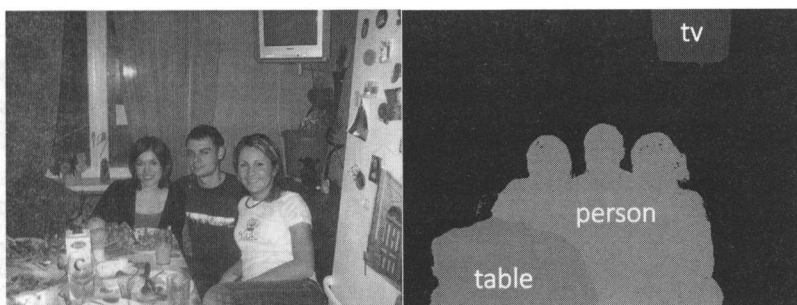


图 1-10 图像分割任务示例（图片来源：Jifeng Dai, Kaiming He, Jian Sun, BoxSup: Exploiting Bounding Boxes to Supervise Convolutional Networks for Semantic Segmentation, arXiv:1503.01640.）



图 1-11 生成模型示意图（图片来源：Andrej Karpathy, Li Fei-Fei, Deep Visual-Semantic Alignments for Generating Image Description, CVPR, 2015.）

除了上面这些应用，深层模型还在很多领域产生了比较大的影响，这里就不再一一介绍这些应用了。

## 1.4 本书的主要内容

在 1.3 节中读者已经看到了许多深度学习的应用，但这只是深度学习这块版图的一部分。实际上，在完成本书的过程中，这块巨大的版图还在以一种惊人的速度迅速扩张。几乎每一天都有新的想法涌现出来，呈现在公众面前。为了能更好地跟上这块版图扩张的速度，读者不仅要了解每天发生的最新进展，而且要从头开始打好基础，把深度学习的基本概念弄清楚，所谓“磨刀不误砍柴工”。

那么基本概念都有哪些呢？在图像领域，最基本的概念是深度卷积神经网络。它由两个最主要的组成部分：经典的全连接层和卷积层。了解了这两个部分，就掌握了卷积神经网络最核心的部分。当读者看到一个深度卷积神经网络，就能马上建立起对这个模型的基本印象。本书第 3 章将介绍全连接层的基本知识。神经网络起初完全以全连

接层为核心组成，后来的深度神经网络（Deep Neural Network，简称 DNN）也以它为主，所以我们有必要详细地了解这部分内容。第 4 章将介绍卷积层的基本知识。作为现在人气远超全连接层的它，在视觉、听觉和自然语言等很多领域都有很广泛的应用，其背后的思想也早已影响了无数人。它是一个非常值得深入研究的模型。

第 5 章将介绍一个经典的深度学习开源框架：Caffe 的使用方法及源代码内容。古语有云：“工欲善其事，必先利其器”。想要了解更多关于深度学习和深层模型的知识，一套顺手好用的工具必不可少。当今，科研界与工业界涌现出了许多优秀的深度学习框架，它们各有特点。在众多框架中，本书选择了一个代码相对简单且易于阅读修改的框架 Caffe，并深入介绍了这个深度学习框架的细节。

第 6 章将介绍卷积神经网络的数值问题。深层模型的数值问题一直是它的一大弱点，由于模型很深，模型内部参数的梯度要经过很多层次的计算和传导才能得到。因此，即使模型在数值上存在一些小问题，经过深层次的传导，这些小问题也会被放大变成大问题。有时，模型里的数字会变得很大导致向上溢出，有时会变得很小导致模型停滞不前，所以掌握基本的分析方法，以及一些常用的参数和数据的设置方法是十分必要的。

第 7 章将介绍卷积神经网络的网络结构。深层模型的一大特点是层数多且排列复杂，这意味着深层模型的网络结构可以有許多变化，有些模型因为无法学习到很强的能力而被人们抛弃，有些模型因为完成了一些有挑战的任务而成为经典。这些经典正是本书想要介绍给读者的。

第 8 章将介绍卷积神经网络的一些优化方法。由于深层模型的网络结构十分复杂，它的优化方法和经典的浅层网络有很大不同，一些在浅层网络中拥有的优秀性质在深层模型中不复存在。因此，深层模型的优化充满了挑战。了解这些优化算法会让模型训练更加得心应手。

第 9 章将介绍卷积神经网络模型的一个应用领域：图像分割。本章将详细介绍一些经典的深层模型，它们很好地解决了图像分割这个细粒度的问题。同时，介绍很多与概率图模型相关的知识，帮助读者更好地了解图像分割算法的全过程。

第 10 章将介绍另一个十分火热的应用方向：生成模型。随着深层模型能力的不断增强，模型能够完成令人惊奇的事情，那就是完成对高度复杂数据的生成式建模。虽然建模过程充满了挑战，但它可以帮助我们完成很多有意义的事情，非常值得研究和探索。

以上就是本书的主要内容。书中的内容并不能完全包含深度学习这些年的发展，同时，书中介绍的内容随着时间的流逝可能会退出舞台，甚至被证明不是最优选择，但是这些内容对读者来说仍然十分重要，如果能对这些内容有更深刻的认识，相信读者能更从容地面对未来涌现出的新的科研成果和工业成就。

## 1.5 总结

本章首先从感性的角度介绍了机器学习的基本形式和主要组成部分,其次阐述了机器学习算法的发展及深度学习爆发的部分原因,最后简单介绍了深度学习在视觉领域的部分应用。从第2章开始,本书将逐步带领读者走进深度学习,开始了解深度学习中方方面的内容。

# 2

## 数学与机器学习基础

在正式开始深度学习的旅程之前，我们还是要对数学和机器学习的基础知识做一些介绍。不同于第1章生动的文字描述，从这一刻开始书中将会出现令人讨厌的数学公式了。物理学界泰斗霍金曾说过：“每多写一个公式就会吓跑一半读者”，但愿这句话不要成为现实。下面的一些内容总体上比较基础，有这方面基础的读者可以跳过本章，想要温习基础知识的读者可以仔细阅读。总而言之，看完了本章，我们在机器学习基础方面将获得很多的共识，阅读本书中后面的章节时将不会被突然出现的名词搞晕。

### 2.1 线性代数基础

线性代数是一门与机器学习紧密相关的数学课程，其中的很多定理、性质和方法论在机器学习中起到了关键性的作用。这一小节旨在简单回顾线性代数中的一些基本内容，为后面的内容做铺垫。实际上这些内容都被涵盖在大学的线性代数课本中，只是大学课本没有专门围绕着机器学习内容进行讲述，下面的内容将围绕着线性变换的核心概念展开。

在线性代数中，大家都接触过“矩阵”这个概念，对于计算机专业的读者来说，矩阵  $A_{m \times n}$  可以看作一个二维数组  $A[m][n]$ ，其中第一维表示行，第二维表示列。矩阵也拥有自己的运算体系，相同维度的矩阵可以相加，矩阵可以和标量相乘，在维度匹配的情况下，矩阵与矩阵之间也可以相乘。矩阵运算满足加法交换律、乘法结合律，不满足乘法交换律。



矩阵运算中经常会遇到一些特殊操作，例如**转置**。矩阵  $A_{m \times n}$  经过转置后就变成了矩阵  $A_{n \times m}^T$ ，相当于矩阵沿着主对角线进行翻转得到了新的矩阵。如果有一个矩阵，它的转置和它本身相同，那么这个矩阵就被称为**对称矩阵**。这类矩阵将成为本书中的一大主角。

矩阵的另一类运算是除法运算。并不是每一个矩阵都可以被除，在线性代数中，如果一个矩阵可以被除，那么就称这个矩阵是**可逆的**。关于矩阵可逆的判断条件还有很多，这里就不一一介绍了。

线性代数中最基本也是最经典的一类运算就是**矩阵和向量的乘法**。如果有矩阵  $A_{m \times n}$  和向量  $x_{n \times 1}$ ，那么这两者是可以相乘的，有：

$$A_{m \times n} \cdot x_{n \times 1} = b_{m \times 1}$$

这个公式实际上和机器学习中的**一些线性模型**相似。如果把矩阵  $A$  想象成一个运算符，那么这个运算符相当于对一个  $n$  维的向量  $x$  做运算，得到一个  $m$  维的向量  $b$ 。如果考虑  $x$  存在于一个  $n$  维的实数向量空间， $b$  存在于一个  $m$  维的实数向量空间，那么  $A$  的作用相当于一个映射，将不同维度的向量关联起来，而且它们之间是线性计算的关系。因此，这种计算也可以被认为是**线性变换**。

线性变换可以从两个角度理解：首先从常规的运算方法来看， $b$  中的每一个元素都是由  $A$  中的一个行向量和  $x$  相乘得到的，它们的关系如图 2-1 所示。

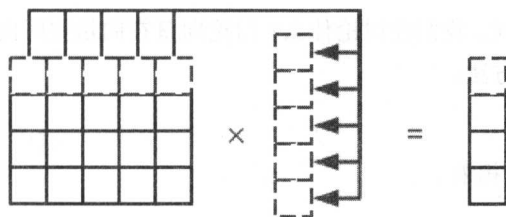


图 2-1 线性变换的一种运算形式

在这种运算形式中， $A$  的每一行可以想象成  $x$  中每一个元素的权重，它们计算的结果汇总成一个数，相当于对  $x$  的元素做加权求和，得到的结果代表了对  $x$  数据的汇总。所以计算中将  $n$  个数据做了  $m$  个汇总。如果再深入一点分析，这些汇总有什么含义呢？可以看出，这里的计算是两个向量的内积运算，内积运算具有自己的运算语意，假设有两个向量  $a$  和  $b$ ，那么它们的内积公式可以写作：

$$a \cdot b = |a||b|\cos\theta$$



其中  $\theta$  表示  $\mathbf{a}$  和  $\mathbf{b}$  在向量空间中的夹角。如果两个向量的长度 (norm) 为 1, 那么两个向量的内积表示了两个向量的某种相关度。如果两个向量共线且同向, 那么内积的结果为 1; 如果两个向量相互正交, 那么内积为 0; 如果两个向量方向相反, 那么内积为 -1。

所以从上面的分析中可以看出, 线性变换相当于对  $\mathbf{x}$  完成了  $m$  次内积运算, 每一次内积运算的内容是在考察  $\mathbf{x}$  与  $\mathbf{A}$  的行向量的相关程度。所以  $\mathbf{b}$  就是  $\mathbf{A}$  与  $\mathbf{x}$  的相关程度的汇总。因此作为一个运算符,  $\mathbf{A}$  的内容就显得十分关键了——它直接决定了“心目中”理想的向量: 像  $\mathbf{A}$  则结果会比较大, 不像结果就会比较小。

另一种理解方式并不是运算时常用的方式, 但却是理解线性代数很重要的一种方式。在这种方式中, 运算过程将被重新组合。

- 1.  $\mathbf{A}$  中的每一个列向量和  $\mathbf{x}$  中的每一个元素依次相乘。
- 2. 相乘后的向量再加和在一起, 得到结果  $\mathbf{b}$ 。

这种计算过程如图 2-2 所示。

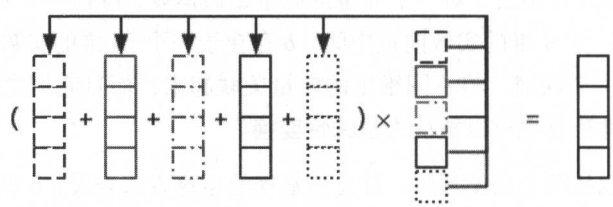


图 2-2 线性变换的另一种解释

在讨论向量的时候, 我们在讨论什么? 讨论向量在向量空间内的运算。向量空间有以下几种经典的运算方法:

- 1. 向量相加。
- 2. 向量与标量数字相乘。
- 3. 向量间求内积。

前面的第一种方法主要使用了第 3 种运算方法, 后面的这种解释方法则主要使用了前面两种运算方法。

在这种解释下,  $\mathbf{x}$  的每一个元素变成了  $\mathbf{A}$  列向量的权重, 实际上这个线性变换变成了向量级别的加权平均。这种解释似乎更为直观。 $\mathbf{A}$  的每一个列向量表示了结果  $\mathbf{b}$  所在空间的一种向量表达, 那么  $\mathbf{x}$  的作用就是平衡这些向量表达并将这些表达揉合成一个新的向量。

从这一种线性变换的角度来分析, 我们就要引出另外一个概念——线性独立。从图 2-2 中读者已经了解到了向量的加权平均, 那么有没有可能经过加权求和得到其中

的某一个向量呢？当然可能。可以让  $\mathbf{x}$  中与这个向量相对应的元素为 1，其他的元素为 0，那么结果就是这个向量。但是如果这里要求这个对应的元素为 0，只能通过其他向量的线性组合（也就是加权求和）来得到这个向量呢？这就不一定了，有些情况下这件事情是可以做到的，有些情况下则无法做到。比方说如下矩阵：

```
a = [
    [1 0 0 0]
    [0 1 0 0]
    [0 0 1 0]
    [0 0 0 1]
]
```

在这个矩阵里，任意一个列向量都不能被其他列向量经过加权求和得到。这样线性独立的概念就比较明确了，如果说一组向量是线性独立的，那么其中的任意一个向量都不能被其他向量通过加权相加得到。

这个性质有什么用处呢？实际上在对线性代数的介绍中很多概念已经被省略，例如矩阵的秩。本节不去考虑那些概念，只是用刚才的运算做进一步的分析。对于  $\mathbf{Ax} = \mathbf{b}$  这样的运算，如果  $\mathbf{A}$  中的列向量组不是线性独立的，那么为了求出结果  $\mathbf{b}$ ，我们可以使用更小维度的  $\mathbf{A}$  和  $\mathbf{x}$ 。比方说现在有如下的运算：

```
import numpy as np
A = np.array([
    [1, 0, 0]
    [0, 1, 0]
    [1, 1, 0]
])
b = np.array([1, 1, 1])
np.dot(A.T, b)
# array([2, 2, 0])
```

如果想要得到同样的结果，我们可以缩小  $\mathbf{A}$  的维度：

```
A = np.array([
    [1, 0, 0]
    [0, 1, 0]
])
b = np.array([2, 2, 0])
np.dot(A.T, b)
# array([2, 2, 0])
```

这说明完成同一个运算时，矩阵  $A$  可以变得更小。上面第一段代码中， $A$  是一个  $3 \times 3$  的矩阵，里面包含着 3 个 3 维的列向量，而第二段代码中， $A$  是一个  $2 \times 3$  的矩阵，里面包含着 3 个 2 维的列向量，这说明上面一段代码中的矩阵  $A$  看似处于 3 维空间中，但实际上它的列向量只存在于其中的一个 2 维子空间中。

线性代数中有一种特殊的线性无关向量组，这个向量组就是基。基在英文中称为“basis”，这里也可以理解成“基础”的含义。最常见的基的形式就是欧式坐标轴体系中，那些坐标轴方向上的向量组成的向量组。基具有以下的特点。

- 它们的组成部分——这些向量是线性无关的。
- 通过这些向量的线性组合，可以表示所在空间内的任意一个向量。比方说对于上面提到的 2 维空间，对于任意一个实数向量，都可以利用这组基通过线性变换表示出来： $(x, y) = x \cdot (1, 0) + y \cdot (0, 1)$

除了线性相关与无关这样的关系，线性代数中还有一种常见的关系，被称为正交。如果一个矩阵和一个向量相乘的结果为 0 向量，而矩阵和向量都不全为 0，那么从线性变换的角度来看，可能有两种情况：

- 向量和矩阵的每一个行向量点乘为 0。
- 矩阵列向量经过被乘向量的线性组合后，相互抵消。

## 2.2 对称矩阵的性质

对称矩阵是线性代数中十分重要的一种矩阵类型，因此这里将它单独列出作为一节进行分析。

### 2.2.1 特征值与特征向量

前面提到了线性变换的运算，那么我们就要深入分析线性变换中的一类特殊的变换。让我们再回到上面的公式：

$$Ax = b$$

此时这个公式对其中的两个向量还没有太多的约束，如果我们对公式增加如下两个约束。

1.  $x$  和  $b$  的维度相同，同为  $n$  维，那么  $A$  就是一个  $n \times n$  的方阵，是  $n$  维空间内的一个算子。

2.  $\mathbf{x}$  和  $\mathbf{b}$  共线，也就是说  $\mathbf{b} = \lambda \mathbf{x}$ 。

这里再将  $\mathbf{x}$  换为这个公式里常用的符号—— $\mathbf{e}$ ，那么上面的公式就可以变为：

$$\mathbf{A}\mathbf{e} = \lambda \mathbf{e}$$

此时我们发现，经过  $\mathbf{A}$  的线性变换后，向量  $\mathbf{e}$  仅仅做了尺度的缩放，而没有做其他的变换，这时我们认为矩阵  $\mathbf{A}$  和向量  $\mathbf{e}$  存在某种特殊的关系，于是人们将向量  $\mathbf{e}$  称为**特征向量**，尺度缩放的系数  $\lambda$  被称为**特征值**。这两个“特征”系的数将成为后面内容介绍的重点。

### 2.2.2 对称矩阵的特征值和特征向量

对称矩阵的特征值和特征向量都有自己的特点。首先，由实数组成的对称矩阵的**特征值全部为实数**，也就是说特征值的共轭数等于自身，以下就来证明。

对于某个实数对称矩阵  $\mathbf{A}$ ，它的共轭矩阵  $\tilde{\mathbf{A}} = \mathbf{A}$ ，对于某个特征值  $\lambda$  和某个非 0 的特征向量  $\mathbf{e}$ ，我们有：

$$\begin{aligned}\tilde{\mathbf{e}}^T \mathbf{A} \mathbf{e} &= \tilde{\mathbf{e}}^T \mathbf{A}^T \mathbf{e} \\ &= (\mathbf{A} \tilde{\mathbf{e}})^T \mathbf{e} \\ &= (\tilde{\mathbf{A}} \tilde{\mathbf{e}})^T \mathbf{e}\end{aligned}$$

所以等式两边有：

$$\begin{aligned}\lambda \tilde{\mathbf{e}}^T \mathbf{e} &= \tilde{\lambda} \tilde{\mathbf{e}}^T \mathbf{e} \\ (\lambda - \tilde{\lambda}) \tilde{\mathbf{e}}^T \mathbf{e} &= 0\end{aligned}$$

因为  $\tilde{\mathbf{e}}^T \mathbf{e}$  不为 0，所以特征值和它的共轭相等，所以对称矩阵的特征值全为实数。

其次，对称矩阵的特征向量相互正交，证明如下。

对于对称矩阵中的两对不同的特征向量、特征值对  $\{\mathbf{e}_1, \lambda_1\}$  和  $\{\mathbf{e}_2, \lambda_2\}$ ，可以推导出：

$$\begin{aligned}\lambda_1 \mathbf{e}_1 \cdot \mathbf{e}_2 &= (\lambda_1 \mathbf{e}_1)^T \mathbf{e}_2 \\ &= (\mathbf{A} \mathbf{e}_1)^T \mathbf{e}_2 \\ &= \mathbf{e}_1^T \mathbf{A}^T \mathbf{e}_2 \\ &= \mathbf{e}_1^T \mathbf{A} \mathbf{e}_2 \\ &= \mathbf{e}_1^T (\lambda_2 \mathbf{e}_2)\end{aligned}$$

所以有  $\lambda_1 \mathbf{e}_1 \mathbf{e}_2 = \lambda_2 \mathbf{e}_1 \mathbf{e}_2$ 。

因为  $\lambda_1 \neq \lambda_2$ ，所以  $\mathbf{e}_1 \mathbf{e}_2 = 0$ 。同理，我们可以推导出所有的特征向量对之间都正交，从而完成证明。

如果将对称矩阵所有特征向量的长度限制为 1（也就是进行标准化），那么这些特征向量组成的矩阵就成了一个标准正交矩阵。这个标准正交矩阵也拥有一个性质，那就是它的逆矩阵等于它的转置矩阵，也就是它自身，由此可得：

$$\begin{aligned} \mathbf{E} \mathbf{E}^T &= \mathbf{I} \\ \mathbf{E}^T &= \mathbf{E}^{-1} = \mathbf{E} \end{aligned}$$

### 2.2.3 对称矩阵的对角化

讲完了上面的两个性质，下面介绍对称矩阵的另一个性质。前面提到了线性变换操作，如果这一次执行线性变换的是一个对称矩阵，那么这个过程会发生什么情况呢？

首先，根据特征值与特征向量的定义，可知：

$$\mathbf{A} \mathbf{e} = \lambda \mathbf{e}$$

将所有的特征值和特征向量融合到一起，并用  $\mathbf{\Lambda}$  表示对角线为特征值，其余位置为 0 的对角阵，那么就有：

$$\mathbf{A} \mathbf{E} = \mathbf{E} \mathbf{\Lambda}$$

2.2.2 节提到了对阵矩阵的特征向量组成了标准正交矩阵，于是有：

$$\mathbf{A} = \mathbf{E} \mathbf{\Lambda} \mathbf{E}^{-1} = \mathbf{E} \mathbf{\Lambda} \mathbf{E}^T$$

这个过程被称为对称矩阵的对角化，那么这个过程有什么作用呢？如果要求矩阵  $\mathbf{A}$  的 2 次幂，对角化的结构可以帮助简化这个过程：

$$\mathbf{A} \mathbf{A} = (\mathbf{E} \mathbf{\Lambda} \mathbf{E}^T)(\mathbf{E} \mathbf{\Lambda} \mathbf{E}^T) = \mathbf{E} \mathbf{\Lambda}^2 \mathbf{E}^T = \mathbf{E} \mathbf{\Lambda}^2 \mathbf{E}$$

于是对于任意次阶乘，都有：

$$\mathbf{A}^n = \mathbf{E} \mathbf{\Lambda}^n \mathbf{E}$$

由于阶乘只发生在对角阵上，于是计算变得简单了许多。这个公式将对后面章节的推导起到帮助作用。

## 2.3 概率论

概率论也是机器学习中十分重要的一部分知识，可以说，当今机器学习的理论基础就是统计和概率论。如果要展开这部分的知识，恐怕一本书也讲不完，这里同样介绍最简单的概念。

### 2.3.1 概率与分布

阿甘的妈妈曾经说过：“Life is like a box of chocolates, you never know what you’re going to get.”（电影《阿甘正传》台词）。生活中充满了不确定性，这些随机给我们带来了快乐，也带来了烦恼。概率论中的很大一部分工作就是描述这些不确定性。最先引出来的概念是**随机事件**，一个随机事件就是一件充满了不确定性的事情，比方说明天下雨这件事。明天可能下雨，也可能不下雨，那么到底下不下雨呢？这就要靠概率来描述了。概率一般用  $P(X)$  来表示，括号里的内容  $X$  就是计算概率的实体。为了度量的方便，概率值的范围被限定在  $[0, 1]$  之间。概率值为 0 表示事件完全不会发生，概率值为 1 表示事件一定会发生。那么随机事件的概率值就可以写作：

$$P(\text{明天下雨}) = 0.4$$

$$P(\text{明天不下雨}) = 0.6$$

实际上上面的写法还是有些不方便，于是前辈又发明了**随机变量**这样的概念，把随机事件抽象成随机变量的一种取值。于是我们有一个随机变量  $X$  表示明天是否下雨， $X=0$  表示不下雨， $X=1$  表示下雨。于是概率又被重新定义为：

$$P(X = 0) = 0.4$$

$$P(X = 1) = 0.6$$

实际上通过这样的定义，概率的表示变成了一种**映射**，它将随机变量的取值和概率值一一对应，成为这个空间中的测度。由于这个测度空间具有很多与欧氏空间相同的良好性质，很多数学分析相关的知识都可以用在这类空间上。如果继续细分，上面这个例子中的随机变量被称为**离散随机变量**。还有一种变量被称为**连续随机变量**，比方说明天下雨的降雨量这个随机变量，它取值的范围是  $[0, +\infty]$ 。这时很多概率相关的运算就需要用上微积分的知识了。

随机变量毕竟是随机变量，我们无法预知它的结果，但是如果已经知道了它的概率分布，猜测它可能的结果还是有可能的，这就引出了猜测它结果的一些描述量，例如期望  $E$  和方差  $\text{Var}$ ：

离散随机变量的期望：

$$E(x) = \sum_x P(x)x$$

连续随机变量的期望：

$$E(x) = \int_x P(x)x\mathrm{d}x$$

离散随机变量的方差：

$$\text{Var}(x) = \sum_x P(x)(x - E(x))^2$$

连续随机变量的方差：

$$\text{Var}(x) = \int_x P(x)(x - E(x))^2\mathrm{d}x$$

在很多场景下，这两个描述量很好地刻画了随机变量的性质，因此它们也被广泛使用。

聊过了单一随机变量的一些定义，下面将进入多随机变量的概率度量刻画问题。对于两个随机变量  $X, Y$ ，它们同时取值的概率被称为**联合概率** $P(X, Y)$ 。当然在实际运算过程中，并不需要两个随机变量在真实世界的同一时刻发生才能知道结果，只需要计算它们同时出现的可能即可，比方说  $X$  表示天要不要下雨， $Y$  表示要不要带伞。那么  $P(X = 1, Y = 1)$  就表示天下雨且带了伞的概率。

之所以强调两个随机变量同时取值，是因为有时两个随机变量的联合概率并不等于两件事情单独发生概率的乘积，即：

$$P(X, Y) \neq P(X) \cdot P(Y)$$

原因在于随机变量之间存在着某种联系。比方说天下了雨，那么人就容易产生带伞的行为，所以两件事情同时发生实际上会包含这层含义。但是把两件事情的概率单独考虑就没有这层意思了，所以对概率的度量就会产生偏差。为了解决这个问题，**条件概率**应运而生。 $P(Y|X)$  表示当  $X$  取值之后  $Y$  取值的概率，所以联合概率正确的展开

方式是这样的：

$$P(X, Y) = P(X) \cdot P(Y|X) = P(Y) \cdot P(X|Y)$$

上面这个公式又可以推导出机器学习界最重要的概率公式（没有之一）——贝叶斯定理：

$$P(Y|X) = \frac{P(X|Y) \cdot P(Y)}{P(X)}$$

这个重要的公式相信读者都有所了解。这个公式中的每一项还有自己的名字， $Y$  被称为隐含变量， $X$  被称为观察变量， $P(Y)$  被称为先验（Prior），它表示我们对一个随机变量概率最初的认识； $P(X|Y)$  被称作似然（Likelihood），它表示在承认先验的条件下另一个与之相关的随机变量的表现； $P(Y|X)$  被称作后验（Posterior），表示当拥有  $X$  这个条件后  $Y$  的概率，由于有  $X$  这个条件，后验概率可能与先验概率不同； $P(X)$  是一个标准化常量（Normalized Constant），由于公式中一般认为  $X$  已知，所以它一般被当成一个常量去看待。

除了计算两个随机变量取值的概率，有些场景还需要计算两个随机变量的相关关系，这个关系可以用协方差表示，它的公式如下所示：

$$\begin{aligned} \text{Cov}(X, Y) &= E[(X - E[X])(Y - E[Y])] \\ &= E[XY] - 2E[Y]E[X] + E[X]E[Y] \\ &= E[XY] - E[X]E[Y] \end{aligned}$$

如果  $X$  和  $Y$  的期望为 0，那么协方差就等于第一项的内容。

上面介绍了随机变量的基本表达形式和基本运算方式，但是对于如何用函数具体表达一个或多个随机变量，这个问题还是充满了挑战。为了让大家方便地分析这些随机变量，找出其中的规律，前辈们发明了很多经典的概率分布，这些概率分布可以当作对同一类问题的模板，满足类似性质的随机变量都可以用这些模板帮助分析。比方说对于离散随机变量，有经典的伯努利分布：

```
import numpy as np
def bornulli(p):
    return 1 if np.random.rand() > p else 0
```

对于连续随机变量，经典的分布有高斯分布：

```
def gaussian(mu, std):
    return np.random.normal(mu, std)
```



相信大家对这些分布都不陌生。一般来说，如果一个随机变量的表现和某个概率分布的具体形式非常接近（几乎处处相等），那就可以认为这个随机变量服从这个概率分布。由于这些概率分布只是一个模板，具体的参数并不固定，因此在说明分布时还要明确分布的参数。

### 2.3.2 最大似然估计

这些分布虽然规定了它们的形式，但是其中仍然包含一些参数，如果读者已经观察到，某个随机变量的采样出来的样本服从某种分布，就可以根据这些样本对分布的参数进行估计。参数估计的方法在很多机器学习算法中也都有用到，其中一类十分经典的算法被称为**最大似然法**。由于每一个样本是否出现都对应着一定的概率，而且一般来说这些样本的出现都不那么偶然，因此我们希望这个概率分布的参数能够以最高的概率产生这些样本。如果观察到的数据为  $D_1, D_2, D_3, \dots, D_N$ ，那么极大似然的目标如下：

$$\max P(D_1, D_2, D_3, \dots, D_N)$$

计算联合概率总归不是一件很容易的事，如果样本数量非常大，那么对联合概率的计算会让人崩溃。所以这里一般会引入一个假设，也就是大家常说的**独立同分布**，**independent and identically distributed (i.i.d.)**。如果每一个样本既属于我们现在要求解的分布，同时它们彼此之间又相互独立，彼此出现的概率互不影响，那么根据条件独立的原则，目标公式就可以变为：

$$\max \prod_i^N P(D_i)$$

看上去计算复杂度降低了许多，对于优化问题，最容易想到的方法就是求导数取极值。如果目标函数是一个凸函数，那么它的导数为 0 的点就是极值点。但是现在的公式是连乘式，求导十分麻烦，这时对数函数就派上了用场。将函数取对数，函数的极值点不会改变，于是公式就变成了：

$$\max \sum_i^N \log P(D_i)$$

变成了这个样子，求导也变得简单了许多，下面的计算也就方便了。

下面将举两个最大似然法计算的例子。首先介绍伯努利分布下随机变量的最大似然计算方法，假设  $P(X = 1) = p$ ， $P(X = 0) = 1 - p$ ，综合起来就有：

$$P(X) = p^X(1-p)^{1-X}$$

如果有一组数据  $D$ ，这一组数据是从这个随机变量中采样得来的，那么就有：

$$\begin{aligned}\max_p \log P(D) &= \max_p \log \prod_i^N P(D_i) \\ &= \max_p \sum_i^N \log P(D_i) \\ &= \max_p \sum_i^N [D_i \log p + (1 - D_i) \log(1 - p)]\end{aligned}$$

对这个式子求导，就有：

$$\nabla_p \max_p \log P(D) = \sum_i^N [D_i \frac{1}{p} + (1 - D_i) \frac{1}{p-1}]$$

令导数为 0，就有：

$$\begin{aligned}\sum_i^N [D_i \frac{1}{p} + (1 - D_i) \frac{1}{p-1}] &= 0 \\ \sum_i^N [D_i(p-1) + (1 - D_i)p] &= 0 \\ \sum_i^N (p - D_i) &= 0 \\ p &= \frac{1}{N} \sum_i^N D_i\end{aligned}$$

这就是伯努利分布下最大似然法求出的结果，结果相当于所有采样值的平均值。

其次是基于高斯分布的最大似然法计算，推导过程也比较类似。有：

$$p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

那么用同样的方式计算，有：

$$\begin{aligned}
\max_p \log P(D) &= \max_p \log \prod_i^N P(D_i) \\
&= \max \sum_i^N \log P(D_i) \\
&= \max \sum_i^N \left[ -\frac{1}{2} \log(2\pi\sigma^2) - \frac{(D_i - \mu)^2}{2\sigma^2} \right] \\
&= \max \left[ -\frac{N}{2} \log(2\pi\sigma^2) - \frac{1}{2\sigma^2} \sum_i^N (D_i - \mu)^2 \right]
\end{aligned}$$

首先对  $\mu$  求导，有：

$$\frac{\partial \max_{\mu} \log P(D)}{\partial \mu} = -\frac{1}{\sigma^2} \sum_i^N (\mu - D_i)$$

令导数为 0，就有：

$$\begin{aligned}
-\frac{1}{\sigma^2} \sum_i^N (\mu - D_i) &= 0 \\
\mu &= \frac{1}{N} \sum_i^N D_i
\end{aligned}$$

其次对  $\sigma^2$  求导，有：

$$\frac{\partial \max_{\sigma^2} \log P(D)}{\partial \sigma^2} = -\frac{N}{2\sigma^2} + \frac{1}{2\sigma^4} \sum_i^N (D_i - \mu)^2$$

令导数为 0，有：

$$\begin{aligned}
-\frac{N}{2\sigma^2} + \frac{1}{2\sigma^4} \sum_i^N (D_i - \mu)^2 &= 0 \\
\sigma^2 &= \frac{1}{N} \sum_i^N (D_i - \mu)^2
\end{aligned}$$

从伯努利分布和高斯分布的最大似然法结果来看，它们最终求得的参数结果和期望方差的计算方式完全一致。

有关概率论的内容就介绍这些，更多内容将随着后面章节的内容介绍。

## 2.4 信息论基础

信息论也是一门和机器学习紧密相关的学科，这里面的一些基本概念，例如熵，读者应该多少有些了解。下面我们就来介绍熵的那些事。

大家都知道这个世界上有很多不确定的事情，比方说明天的天气，足球世界杯哪支队伍摘得桂冠。如果有人能将这些不确定的事情确定下来，或者提前告知人们它未来的状态，那么很显然这个人的能力是非常强大的。这里可以把这种描述不确定性事物的断言称为一种信息，事实上，描述任何一种不确定的事物都可能产生信息，那么这些信息的价值有没有什么差别呢？很多谍战片都在讲述地下特工如何冒生命危险将一个机密情报传递给后方的军队，可见这个情报是非常有价值的，而敌方军队是否会不战而降呢？这个问题似乎不用考虑，大家都能给出结论。

所以可见不同的信息拥有不同的价值，那么它的价值该如何体现呢？从信息论的角度来说，如果一个信息的随机性越大，那么把它确定下来的价值也就越大。例如，我们有一个正常的硬币，一面是字，另一面是花，将硬币抛在空中然后落下，朝上的那一面会是哪一面呢？由于这里提到的是一个正常的硬币，于是可以认为每一面朝上的概率是相同的，所以每一面朝上的概率都是 50%；如果另一枚硬币两面都是字，那么字的一面朝上的概率就是 100%。所以猜中第二枚硬币朝上的那面比第一枚要容易得多。

虽然定性地猜测“哪枚硬币朝上一面的信息量大”这件事并不难，但是这个世界上有许多类似的事情，如何从信息论的角度比较任意两个随机变量的信息价值呢？这就需要一种定量的分析方法。于是前辈发明了“熵”这个概念。

首先来看看离散随机变量的“熵”。有些著作中将“熵”形容为了解真相的“惊喜度”。如果一个随机事件发生的概率为 100%，那么它的发生对我们来说毫无惊喜可言——因为它肯定会发生；而如果它发生的概率为百万分之一，那么它的发生一定会让人惊喜不已。可以想象百万分之一概率发生的都是什么事件——比方说彩票中奖，这样的事件让人惊喜的可能性当然非常大。

把一个离散随机变量拆解成有限个或者无限个随机事件，分别把这些事件的“惊喜度”计算出来，然后把惊喜度结合起来，就是这个随机变量的惊喜度了。看上去这里需要对“惊喜度”做更加具体的定义。同其他一些公理性质的概念一样，“惊喜度”的计算公式不会被直接给出，给出的是“惊喜度”公式应该具备的一些性质，这一点和勒贝格测度的定义方法有些类似。令这个“惊喜度”的计算公式为  $f(p(x))$ ，其中的输入  $p(x)$  为某个随机事件发生的概率， $p(x) \in [0, 1]$ 。它需要满足下面的性质。

1.  $f(1) = 0$ 。一定发生的事情没有“惊喜”。

2. 如果  $p(x) < p(y)$ , 那么  $f(p(x)) > f(p(y))$ 。换句话说一个事情的概率越小, 它的“惊喜度”越大。
3. 函数  $f$  在输入空间是连续的。
4. 两个事件同时发生的“惊喜度”等于各自事件发生的“惊喜度”的和, 由于两个事件是互不相交的, 所以这条定理和测度的次可加性性质很像。
5.  $f(0.5) = 1$ , 这个性质主要是做归一化用, 就像外测度对测度定义的限制一样。

那么什么样的函数能同时满足这些条件呢? 科研人员最终找到了一个函数:  $f(x) = -\log_2 x$ 。它完美地满足所有的性质。最后把随机事件发生的概率当作这个事件的权重, 把所有事件的“惊喜度”作加权平均, 这样就得到了随机变量整体的“惊喜度”:

$$\text{entropy} = - \sum_x p(x) \log p(x)$$

同样地, 熵的定义也可以推广到连续变量上:

$$\text{entropy} = - \int_x p(x) \log p(x) dx$$

这个公式虽然满足上面的那些性质, 但是看上去总是不够直观, 回到刚才那个硬币的例子, 假设手里有一枚硬币, 硬币有字一面朝上的概率为  $p$ , 所以花一面朝上的概率为  $1-p$ 。知道这些就可以写出求这个问题熵的公式:

$$\text{entropy}(p) = -p \log p - (1-p) \log(1-p)$$

这个公式的图像可以通过下面的代码生成:

```
import numpy as np
x = np.linspace(0,1,101)
y = -x * np.log2(x) - (1-x) * np.log2(1-x)
y[np.isnan(y)]=0
plt.plot(x,y)
plt.show()
```

结果如图 2-3 所示。

当两个面朝上的概率相同时, 它的熵最大。这也和大家的直觉相符, 正是因为两个事件的概率相同, 我们才难以判断最终结果, 这时知道结果的“总体惊喜度”才会最大。后面的章节还会介绍其他与信息论相关的概念, 到时再细讲。

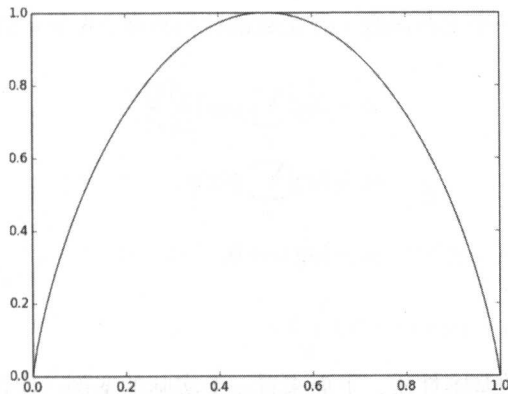


图 2-3 硬币问题的概率与熵的关系图

## 2.5 KL 散度

**KL 散度**是概率论和信息论中十分重要的一个概念。它是描述两个概率分布  $P$  和  $Q$  差异的一种方法。

离散概率分布的 KL 散度计算公式为：

$$\text{KL}(p||q) = \sum p(x) \log \frac{p(x)}{q(x)}$$

连续概率分布的 KL 散度计算公式为：

$$\text{KL}(p||q) = \int p(x) \log \frac{p(x)}{q(x)} dx$$

在后面章节要提到的 Variational Inference 中，我们希望能够找到一个相对简单好算的概率分布  $q$ ，使它尽可能地近似待分析的后验概率  $p(z|x)$ ，其中  $z$  是隐变量 (Hidden Variable)， $x$  是观测变量 (Observed Variable)。在这里目标函数就是 KL 散度，它可以很好地测量两个概率分布之间的距离。两个分布越接近，那么 KL 散度越小；如果越远，KL 散度就会越大。

KL 散度的结果是非负的，这里可以简单证明得出：

$$\begin{aligned} \text{KL}(p||q) &= \sum_x p(x) \log \frac{p(x)}{q(x)} \\ &= - \sum_x p(x) \log \frac{q(x)}{p(x)} \end{aligned}$$

由于对数函数是一个上凸函数（凸函数的性质将在 2.6 节介绍），所以有：

$$\begin{aligned}
 &\geq -\log\left[\sum_x p(x) \frac{q(x)}{p(x)}\right] \\
 &= -\log\left[\sum_x q(x)\right] \\
 &= -\log 1 = 0
 \end{aligned}$$

由此得证。

看完了 KL 散度的基本性质，下面来看一个实际的案例：假设有两个随机变量  $x_1$  和  $x_2$ ，各自服从一个高斯分布  $N_1(\mu_1, \sigma_1^2)$ ,  $N_2(\mu_2, \sigma_2^2)$ ，那么这两个分布的 KL 散度该怎么计算呢？

在前面的 2.3 节中，读者知道了高斯分布的概率密度函数：

$$N(\mu, \sigma) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

那么  $KL(p_1||p_2)$  就等于

$$\begin{aligned}
 &\int p_1(x) \log \frac{p_1(x)}{p_2(x)} dx \\
 &= \int p_1(x) (\log p_1(x) - \log p_2(x)) dx \\
 &= \int p_1(x) \left( \log \frac{1}{\sqrt{2\pi\sigma_1^2}} e^{-\frac{(x-\mu_1)^2}{2\sigma_1^2}} - \log \frac{1}{\sqrt{2\pi\sigma_2^2}} e^{-\frac{(x-\mu_2)^2}{2\sigma_2^2}} \right) dx \\
 &= \int p_1(x) \left( -\frac{1}{2} \log 2\pi - \log \sigma_1 - \frac{(x-\mu_1)^2}{2\sigma_1^2} + \frac{1}{2} \log 2\pi + \log \sigma_2 + \frac{(x-\mu_2)^2}{2\sigma_2^2} \right) dx \\
 &= \int p_1(x) \left( \log \frac{\sigma_2}{\sigma_1} + \left[ \frac{(x-\mu_2)^2}{2\sigma_2^2} - \frac{(x-\mu_1)^2}{2\sigma_1^2} \right] \right) dx \\
 &= \int \left( \log \frac{\sigma_2}{\sigma_1} \right) p_1(x) dx + \int \left( \frac{(x-\mu_2)^2}{2\sigma_2^2} \right) p_1(x) dx - \int \left( \frac{(x-\mu_1)^2}{2\sigma_1^2} \right) p_1(x) dx \\
 &= \log \frac{\sigma_2}{\sigma_1} + \frac{1}{2\sigma_2^2} \int ((x-\mu_2)^2) p_1(x) dx - \frac{1}{2\sigma_1^2} \int ((x-\mu_1)^2) p_1(x) dx
 \end{aligned}$$

到这一步，读者可以发现右边最后一项可以化简掉，这个积分符号里面的东西是不是看着很熟悉？没错，它就是连续变量方差的计算公式，于是经过约分，这一项就变成了  $\frac{1}{2}$ 。继续推导，有：

$$\begin{aligned}
&= \log \frac{\sigma_2}{\sigma_1} + \frac{1}{2\sigma_2^2} \int ((x - \mu_2)^2) p_1(x) dx - \frac{1}{2} \\
&= \log \frac{\sigma_2}{\sigma_1} + \frac{1}{2\sigma_2^2} \int ((x - \mu_1 + \mu_1 - \mu_2)^2) p_1(x) dx - \frac{1}{2} \\
&= \log \frac{\sigma_2}{\sigma_1} + \frac{1}{2\sigma_2^2} \left[ \int (x - \mu_1)^2 p_1(x) dx + \int (\mu_1 - \mu_2)^2 p_1(x) dx \right. \\
&\quad \left. + 2 \int (x - \mu_1)(\mu_1 - \mu_2) p_1(x) dx \right] - \frac{1}{2} \\
&= \log \frac{\sigma_2}{\sigma_1} + \frac{1}{2\sigma_2^2} \left[ \int (x - \mu_1)^2 p_1(x) dx + (\mu_1 - \mu_2)^2 \right] - \frac{1}{2} \\
&= \log \frac{\sigma_2}{\sigma_1} + \frac{\sigma_1^2 + (\mu_1 - \mu_2)^2}{2\sigma_2^2} - \frac{1}{2}
\end{aligned}$$

经过大段让人发蒙的公式推导，最终的结论已经得出。假设  $N_2$  是一个正态分布，也就是说  $\mu_2 = 0, \sigma_2^2 = 1$ ，那么  $N_1$  长成什么样子能够让 KL 散度尽可能地小呢？

将  $N_1$  中的变量代入，公式变成：

$$\text{KL}(\mu_1, \sigma_1) = -\log \sigma_1 + \frac{\sigma_1^2 + \mu_1^2}{2} - \frac{1}{2}$$

读者简单分析一下就能猜测到，当  $\mu_1 = 0, \sigma_1 = 1$  时，KL 散度最小。从公式中可以看出，如果  $\mu_1$  偏离了 0，那么 KL 散度一定会变大。而  $\sigma_1$  变化对 KL 散度的影响则不那么明显：

- 当  $\sigma_1$  大于 1 时， $\frac{1}{2}\sigma_1^2$  将越变越大，而  $-\log \sigma_1$  越变越小。
- 当  $\sigma_1$  小于 1 时， $\frac{1}{2}\sigma_1^2$  将越变越小，而  $-\log \sigma_1$  越变越大。

那么，哪边的力量更强大呢？由于公式比较简单，这里就将它的图像画出来进行比较，如图 2-4 所示。

```

import matplotlib.pyplot as plt
import numpy as np
from mpl_toolkits.mplot3d import Axes3D

fig = plt.figure()
ax = Axes3D(fig)
x = np.linspace(0.1, 2, 31)
y = np.linspace(-2, 2, 31)
X, Y = np.meshgrid(x, y)

```



```
Z = -np.log(X)+X*X+Y*Y/2-0.5
ax.plot_surface(X,Y,Z,rstride=1,cstride=1,cmap='rainbow')
plt.show()
```

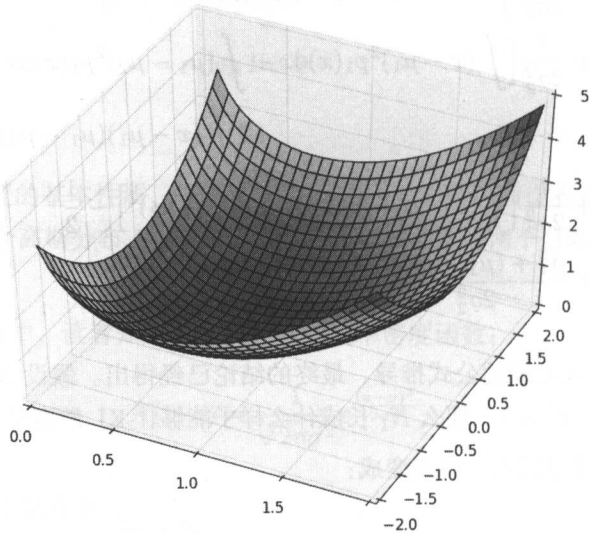


图 2-4 两个高斯分布的 KL 散度图

从图中可以看出二次项的威力更大，函数一直保持为非负，这和前面推导的结论（KL 散度非负的结论）是完全一致的。

完成了 1 维高斯分布间的 KL 散度计算，那么多维高斯分布的 KL 散度是什么样子？首先给出多维高斯分布的公式：

$$p(x_1, x_2, \dots, x_n) = \frac{1}{\sqrt{2\pi \cdot \det(\Sigma)}} e^{(-\frac{1}{2}(x-\mu)^T \Sigma^{-1}(x-\mu))}$$

由于这次是多维变量，里面的大多数计算都变成了向量、矩阵之间的计算。多维高斯分布通常假设各维之间相互独立，因此协方差矩阵实际上是个对角阵。

多维高斯分布间的 KL 散度公式如下所示：

$$KL(p1||p2) = \frac{1}{2}[\log \frac{\det(\Sigma_2)}{\det(\Sigma_1)} - d + \text{tr}(\Sigma_2^{-1}\Sigma_1) + (\mu_2 - \mu_1)^T \Sigma_2^{-1}(\mu_2 - \mu_1)]$$

这个公式将在 10.1 节介绍 VAE 时发挥很重要的作用，感兴趣的读者可以试着推导一下。

## 2.6 凸函数及其性质

凸函数是机器学习中经常见到的一种形式。它拥有非常好的性质，在计算上拥有更多的便利。这种感觉就好像看到一个复杂的 7 次函数  $f(x) = ax^7 + bx + c$ ，但实际上它的 7 次项系数  $a = 0$ ，这个披着复杂函数外皮的家伙其实很好分析。虽然后面介绍的深度学习网络的优化曲面并不具备凸函数的性质，但是凸函数还是作为一个十分经典的模型被大家不断研究学习，更何况它也是众多优化算法的基础，因此下面开始介绍这类函数及其相关特点。

要介绍凸函数，首先要介绍凸集。如果一个集合  $C$  被称为凸集，那么这个集合中的任意两点间的线段仍然包含在集合中，如果用形式化的方法描述，那么对于任意两个点  $x_1, x_2 \in C$ ，和任意一个处于  $[0, 1]$  之间的实数  $\theta$ ，都有：

$$\theta x_1 + (1 - \theta)x_2 \in C$$

下面给出凸函数和非凸函数的对比图象，如图 2-5 所示。可以看出，对于左边的凸集区域，任意两点间的线段都在集合中，而对于右边的非凸集区域，我们可以找到两点间的一条线段，使得线段上的点在集合外。

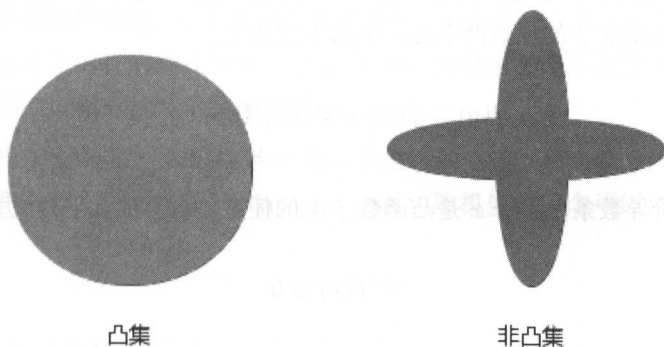


图 2-5 凸集和非凸集

凸函数的定义域就是一个凸集，除此之外，它具备另外一个性质：给定函数中任意两点  $x, y$ ，和任意一个处于  $[0, 1]$  之间的实数  $\theta$ ，有：

$$f(\theta x + (1 - \theta)y) \leq \theta f(x) + (1 - \theta)f(y)$$

这里给出一个凸函数的例子： $f(x) = x^2$ ，然后看看这个性质在函数上的表现，它的图像如图 2-6 所示，图中的横截线代表不等式右边的内容，横截线下方的曲线代表不

等式左边的内容，从图中确实可以看出不等式所表达的含义，如果一个凸函数像  $x^2$  这样是一个严格的凸函数，那么实际上除非  $\theta$  等于 0 或者 1，否则等号不会成立。

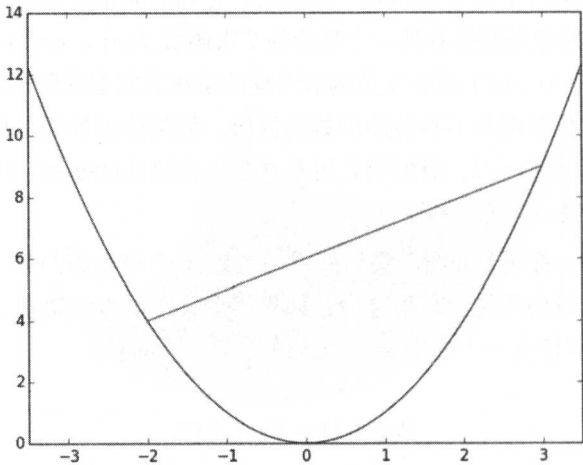


图 2-6 凸函数性质展示

凸函数满足的这个不等式也被称为 **Jensen 不等式**，如果要判定一个函数是不是凸函数，除了用 Jensen 不等式来判定，还可以用下面两种方式判定。首先是一阶导数条件。令  $x, y$  是凸函数  $f$  的任意两个点，那么下式成立：

$$f(y) \geq f(x) + \nabla f(x)^T (y - x)$$

其次是二阶导数条件：令  $x$  是凸函数  $f$  上的任意一点，那么下式成立：

$$\nabla^2 f(x) \geq 0$$

这两个条件是凸函数成立的充要条件，同时也揭示了凸函数的重要性质。从直观的角度分析，对于前面提到的凸函数  $f(x) = x^2$ ，这两个性质显然成立：

$$\begin{aligned} f(y) &= y^2 \geq x^2 + 2x(y - x) \\ &= x^2 + 2x(y - x) + (y - x)^2 - (y - x)^2 \\ &= [x + (y - x)]^2 - (y - x)^2 \\ &= y^2 - (y - x)^2 \\ \nabla^2 f(x) &= 2 \geq 0 \end{aligned}$$

凸函数究竟有什么良好的性质，值得读者花大力研究呢？

如果一个点是凸函数的局部最优值，那么这个点就是函数的全局最优值。更进一步，如果这个函数是强凸函数，那么这个点是函数唯一的全局最优值。这个问题可以通过一阶条件证明，当一个点  $x$  是凸函数的局部最优值时，它的导数为 0，那么对于任意的点  $y$ ，都有

$$f(y) \geq f(x) + \nabla f(x)^T(y - x) = f(x)$$

当函数为强凸函数时，大于等于号将变为大于号。

有了这个性质，我们就可以放心大胆地做函数的优化了。只要找到一个导数为 0 的局部最优值，我们就找到了全局最优值。凸函数相关的理论还有很多，本节就介绍到这里。

## 2.7 机器学习基本概念

在经典的机器学习入门书籍中，读者一般会了解到很多经典的机器学习模型，由于本书介绍的问题主要是监督学习，所以主要介绍一些监督学习相关的内容。在第 1 章，读者已经对模型这个概念有了一定的了解，也了解了模型和函数的关系。在很多场景下，模型都是由函数构成的，函数的形式有很多种，有的比较复杂，有的比较简单。下面假设有为模型准备好作为输入的特征数据  $X$  和作为输出的结果数据  $Y$ ，针对不同的任务，监督学习又可以分成几个具体的类别。

首先是输出  $Y$  的形式，如果结果  $Y$  是一个连续的变量，那么这个问题通常被称为回归问题。这里可以用 Python 的 `scikit-learn` 工具包随机生成一个可以可视化的回归问题数据集。

```
import numpy
import sklearn.datasets as d
import matplotlib.pyplot as plt
reg_data = d.make_regression(100, 1, 1, 1, 1.0)
plt.plot(reg_data[0], reg_data[1])
plt.show()
```

这个例子相对来说十分简单，它的结果也十分直观，输入数据和输出数据都是 1 维的，它们的关系如图 2-7 所示。

如果输出  $Y$  的形式是离散的变量，比方说上面提到的手写数字分类问题，那么这个问题就是分类问题。这里也给出一个简单的分类问题的数据集。

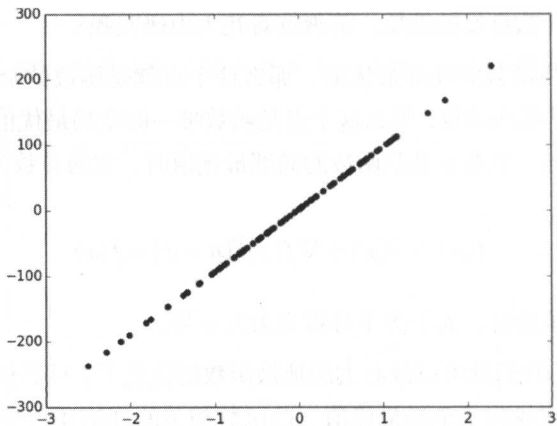


图 2-7 一个回归问题的数据展示

```
import numpy
import sklearn.datasets as d
import matplotlib.pyplot as plt
cls_data = d.make_classification(100, 2, 2, 0, 0, 2)
```

这个例子对应的图像也十分简单，如图 2-8 所示。

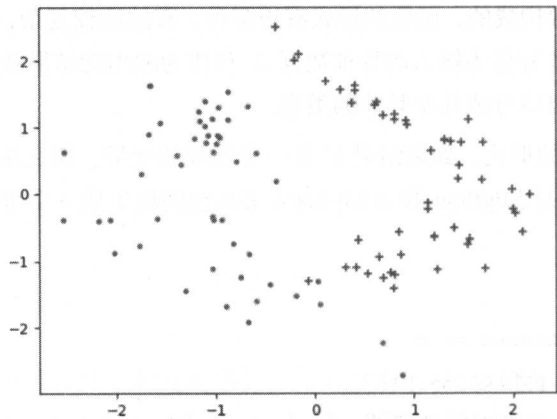


图 2-8 分类问题数据示例

分类问题的目标就是让模型把点号和加号的点区分开。

上面看过了问题的形式，下面来看看问题的解决方法。这里以回归问题为例，为了解决问题，我们可以建立一个模型，让模型充当输入和输出的桥梁。例如，对于上面的回归问题，可以建立线性模型：

$$y = wx + b$$

其中  $x$ ,  $y$  是分别是模型的输入和输出,  $w$  和  $b$  是模型的两个参数, 这两个参数在模型学习训练前是不固定的, 经过训练后, 它才会确定下来。

第一步明确了模型, 下面就来看看模型的目标函数。这里采用平方损失函数, 如果说数据的真实输出为  $t$ , 模型的输出为  $y$ , 那么损失函数定义为:

$$\text{Loss} = \frac{1}{2}(t - y)^2$$

平方损失函数很好地度量了真实输出和模型输出之间的距离。如果模型输出和真实输出差距较大, 那么模型损失函数的结果也较大, 说明了模型还有很大的改进空间。从损失函数的定义来看, 损失函数的最小值为 0, 这就是模型要努力的方向。所以, 最终函数的目标是 minimized 损失函数:

$$\min_{w,b} \text{Loss} = \min_{w,b} \frac{1}{2}(t - y)^2$$

完成了模型和目标函数的定义, 下面就该看看模型该如何学习了。对于这个简单的问题, 直接求出它的闭式解是完全可行的。但在大多数真实场景中, 闭式求解模型最优值是不现实的, 因为那将花费非常长的时间。通常, 我们用基于梯度的方式采用优化的方法求解, 因此这个过程也被称作优化。优化的基本流程分成三个步骤。

1. 为参数设定一些初始值。
2. 利用梯度信息计算参数的更新量。
3. 判断参数优化是否完成, 如果完成, 则停止优化, 否则回到第 2 步。

这三个步骤中, 第 3 步的内容相对简单, 一般有以下几种判断方法。

1. 判断参数所在位置的梯度是否靠近 0 (依梯度收敛)。
2. 判断参数的更新量是否靠近 0 (柯西列收敛)。
3. 判断预设的迭代轮数是否已经完成。

由于机器学习中的模型一般是定义良好的函数, 因此这一部分受到的关注比较小。此外, 对于凸函数优化来说, 第 1 步的内容也比较简单, 优化的重头戏全部在第 2 步中。

对于深度学习来说, 由于种种原因, 越来越多的人发现了第 1 步的重要性。本书的第 6 章将介绍与第 1 步有关的内容, 而深层模型优化的详细内容将在第 8 章介绍。

本节只涉及机器学习中的这几个概念, 更多的概念将在后面章节中慢慢介绍。通过本节, 希望读者至少对模型、目标函数和优化这三个概念有一些了解。

## 2.8 机器学习的目标函数

前面介绍了机器学习中常见的组成部分，下面要花一点时间看看损失函数这部分的内容。常见的损失函数有平方损失函数和交叉熵损失函数，这两种损失函数有什么区别和联系呢？

平方损失函数是一种比较容易理解的损失函数，它直接衡量了模型输出和标准答案在数值上的差距。定义模型输出的向量为  $\mathbf{y}$ ，训练数据的标签向量为  $\mathbf{t}$ ，那么平方损失函数的定义为：

$$\text{SquareLoss} = \frac{1}{2}(\mathbf{y} - \mathbf{t})^2$$

交叉熵的定义就没那么直接了，它需要用到 2.4 节中信息论的知识。这里简单回顾一下，熵这个概念衡量了一个随机变量带来的“惊喜度”，它通过计算每一个取值的惊喜度汇总得到。实际上，这个公式的计算涉及两个部分——惊喜度 + 汇总。如果惊喜度的计算使用一个概率分布，而汇总使用另一个概率分布，那么结果会变成什么样子？这就是交叉熵要表达的内容。

举一个简单的例子，有两个服从伯努利分布的随机变量  $P$  和  $Q$ ，那么它们的交叉熵  $H(P, Q)$  为：

$$H(P, Q) = -P(0) \log Q(0) - (1 - P(0)) \log(1 - Q(0))$$

由于这个公式只有 2 个参数，于是这里就可以将它的图像画出来：

```
import matplotlib.pyplot as plt
import numpy as np
from mpl_toolkits.mplot3d import Axes3D

fig = plt.figure()
ax = Axes3D(fig)
X = np.linspace(0.01, 0.99, 101)
Y = np.linspace(0.01, 0.99, 101)
X, Y = np.meshgrid(X, Y)
Z = -X * np.log2(Y) - (1-X) * np.log2(1 - Y)
ax.plot_surface(X, Y, Z, rstride=1, cstride=1, cmap='rainbow')
plt.show()
```

生成的图像如图 2-9 所示。

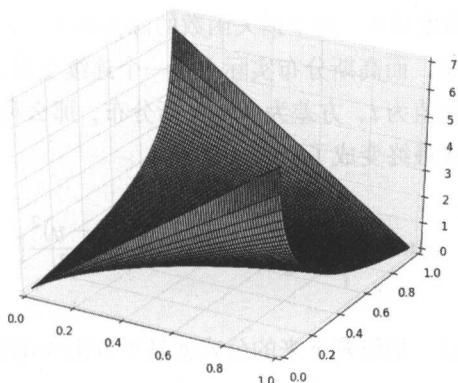


图 2-9 伯努利分布下两个随机变量的交叉熵

当两个分布的取值完全相同时，交叉熵的取值最小。对于损失函数来说，当然是希望模型输出和数据标签相同，因此损失函数的目标就是让交叉熵尽可能地变小。在这一点上函数和目标是一致的。

看上去两个损失函数的目标是差不多的，那么该如何应用它们呢？一般来说，如果最终输出的结果是回归问题的一个连续型变量，使用平方损失函数更合适；如果最终输出是分类问题的一个离散 One-Hot 向量，那么交叉熵损失函数更合适。

但是，看上去平方损失函数既可以做回归问题的损失函数，又可以做分类问题的损失函数，因为它们都可以用距离来衡量，那么平方损失函数在做分类问题的损失函数时有什么弱点呢？

这个问题可以从两个方面回答。一方面，直观上看，平方损失函数对每一个输出结果都十分看重，而交叉熵损失函数只对正确分类的结果看重。假设我们遇到了一个三分类问题，模型的输出自然是一个 3 维的实数向量  $(a, b, c)$ ，向量的每一个元素都表示了对这个类别的预测概率，假设数据的真实结果为  $(1, 0, 0)$ ，那么两个损失函数的公式为：

$$\text{SquareLoss} = (a - 1)^2 + (b - 0)^2 + (c - 0)^2 = (a - 1)^2 + b^2 + c^2$$

$$\text{CrossEntropyLoss} = -1 \times \log a - 0 \times \log b - 0 \times \log c = -\log a$$

可以看出，平方损失函数考虑的内容实际上比交叉熵损失函数多。也就是说，在模型优化的过程中，交叉熵损失的梯度只和正确分类的预测结果有关，而平方损失的梯度还和错误分类有关。除了让正确分类尽可能变大，平方损失函数还会让错误分类都变得更平均，但实际中后面这个调整是不必要的，所以平方损失实际上完成了额外的工作，因此它的实用性就显得没那么强了。但是话说回来，这个考虑在回归问题上就显得非常重要了，因此在回归问题上交叉熵损失显然就不合适了。



另一方面，从理论角度分析，两个损失函数的源头是不一样的。平方损失函数假设最终结果都服从高斯分布，而高斯分布实际上是一个连续变量，并不是一个离散变量。如果假设结果变量服从均值为  $t$ ，方差为  $\sigma$  的高斯分布，那么利用最大似然法就可以优化它的负对数似然，公式最终变成了：

$$= \max \sum_i^N \left[ -\frac{1}{2} \log(2\pi\sigma^2) - \frac{(t_i - y)^2}{2\sigma^2} \right]$$

除去与  $y$  无关的项目，最终剩下来的公式就是平方损失函数的形式。

## 2.9 总结

本章和读者一同回顾了机器学习中常用的数学基础知识。

- 线性代数基础：矩阵、线性变换、线性相关、特征值与特征向量。
- 线性代数中的对阵矩阵。
- 概率论基础：概率、贝叶斯公式、最大似然法估计。
- 信息论基础：熵、KL 散度。
- 凸函数：凸函数的性质。
- 机器学习的基本概念：分类回归模型、目标函数、优化。

# 3

## CNN 的基石：全连接层

全连接层是神经网络中的重要组成部分之一，它主要由两个子部分组成。

- 线性运算部分（以下简称线性部分）：完成线性变换的工作，将输入经过线性变换转换成输出。在下面的介绍中，输入一般用  $X$  表示，输出一般用  $Z$  表示。
- 非线性运算部分（以下简称非线性部分）：紧接着线性部分，完成非线性变换。在下面的介绍中，输入用线性部分的输出  $Z$  表示，输出用线性部分的输入  $X$  表示。

在很多第三方的开源框架中，这两个部分是分开描述定义的；在有些框架中，它们是合在一起的，但一般来说，在分析框架时，两个部分会放在一起考虑。从上面的描述来看，前一层的输出可以用作下一层的输入，而下一层的输出又可以用作下下层的输入，这样一个神经网络就可以由这些全连接层堆叠起来，形成一个庞大的网络模型。

本章先分别介绍这两部分的具体内容：它们的运算的形式、运算语意等，为读者构建一个基本的概念，然后通过实验的方式展示全连接层模型复杂的一面，让读者建立一个对全连接层更感性的认识。最后介绍全连接层优化求解的方法——反向传播法，以及实践过程中应该注意的一些问题。

### 3.1 线性部分

线性部分从运算过程上看就是线性变换，对于一个输入向量  $\mathbf{x} = [x_0, x_1, \dots, x_n]^T$ ，线性部分的输出向量是  $\mathbf{z} = [z_0, z_1, z_2, \dots, z_m]^T$ ，那么线性部分的参数就是一个  $m \times n$  的矩阵  $W$ ，有时再加上一个偏置项  $\mathbf{b} = [b_0, \dots, b_m]^T$ ，于是运算的公式就是下面这个：

$$W \cdot x + b = z$$

从表面上看就是矩阵和向量相乘，再加上一个向量就得到了结果。但读者肯定不满足于这样简单的描述，那就来进一步观察，看看线性部分究竟做了什么事情。用通俗不那么严谨的话来说，线性部分通过数值计算的方式从不同“角度”对输入数据进行分析汇总，得出在这些“角度”下对输入数据的“总结和判断”。这种说法实在有点抽象，下面举一个实际的例子来详细解释这个说法。这里选取卷积神经网络的入门级案例，也是贯穿本书的一个重要的数据集——MNIST 手写数字数据集举例，介绍什么是“总结和判断”。

有些读者可能对 MNIST 数据集不太了解。它是一个用于识别手写数字的训练集合，其中包含了 70000 张手写数字的图像，其中 60000 张被分在训练数据中，剩下的 10000 张分配在测试数据中。每张图像都是一张  $28 \times 28$  的灰度图，内容是 0~9 这十个数字。图像数据显示出来如图 3-1 所示。

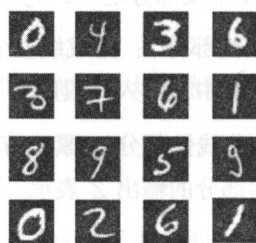


图 3-1 MNIST 数据图像

第 1 章曾经介绍过，经过一定的学习和训练的人类可以轻松认出上面的数字，但是如果一定要理性地分析为什么这些数字会被认出，就有点难以回答。对人类来说这是明摆着的，难道还需要什么理由吗？没错，在人类看来确实非常明显，人类早已习惯了把一个数字图像当成一个整体，利用多年训练的视觉系统在无意识中快速完成识别，但对以理智和逻辑严密著称的计算机来说会非常抓狂——这一点都不明显。如果站在计算机的角度看这些图就会发现这些像素点看上去过于抽象，计算机必须从一堆数字中发现其中的规律，并有理有据地表现出来。计算机眼中的图像如图 3-2 所示。为了做到这一点，它恐怕要费很大的功夫。那么如何找到其中的规律呢？

让我们从最直观的思维方式出发寻找规律，并一步步深入下去。既然每一张图像的大小是固定的，那能否根据某一个位置的像素值判断像素值和最终识别的数字究竟是否有关系呢？例如，图像第 4 行第 7 列的像素值其实就可以用于判断一个手写数字是 6 而不是 7？

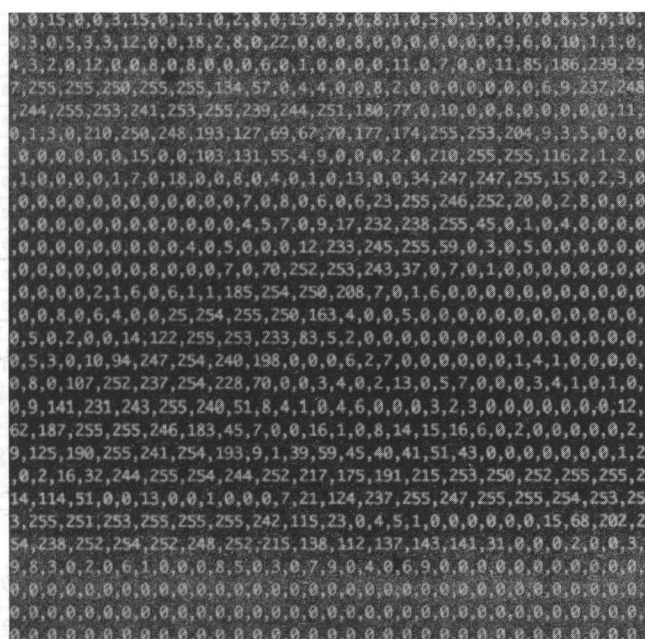


图 3-2 计算机眼中的“图像”

听上去好像不太可能。一个像素的像素值当然不足以帮助我们判断数字的内容，其实把图像所有的像素完全独立地拿出来看，人类也很难知道这个数字是几——因为即使是同一个人书写同一个数字所产生的图像，在某一个位置上的像素值也会有大有小，更何况千人千面，有庞大的人群书写，数字书写的形式会更加千奇百怪，有的写得很大也有的写得很小；有的写得很正，有的写得很歪；有的笔迹粗，有的笔迹细，单从一个像素的特征实在不容易看出来。

但如果把一堆像素点放在一起，观察它们之间的关系，似乎就可以看出一些特征规律。例如，对于一张书写了数字 0 的黑底白字的图像，一般来说，图像正中间部分的像素都是黑色的，而其周围的像素是白色的，这圈白色再往外边又变成了黑色。利用了大量像素的信息，这段描述听上去像是个比较靠谱的特点了，但是这样的描述并不具备很强的操作性，如何把上面这段话转变成数学的形式呢？

一种简单的操作方法如下：首先在图像中心划定一个大概的范围，定义这部分像素就是“中心区域”，并让“中心区域”的每一个像素值与 0 相乘，然后让“中心区域”以外一段区域的每一个像素值与正数相乘，接着让再外面区域的每一个像素值与 0 相乘，最后把这些相乘的结果加和在一起，得到一个汇总数字。从直觉上理解，这个最终的结果越大，越表示我们的这张图像可能是“0”这个数字，反之则不是。这段充满数学味道的描述可以用图 3-3 表示。

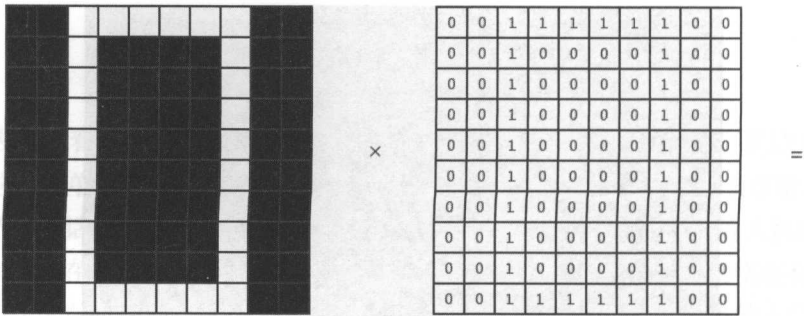


图 3-3 图像 0 的特征描述（左图是一个背景色为黑色、前景色为白色的图像，图像的大小为 10×10，图像中的数字为 0。右图是与描述对应的一组参数，将图像的像素和参数一一对应相乘，就可以得到最终的结果。由于在灰度色彩空间下，白色像素的像素值为 255，黑色像素的像素值为 0，所以当前的图像与参数相乘会得到一个相对比较大的数值）

刚才提到的这个特征只描述了“一个图像像 0”这件事情，除此之外，图片还需要被检测长得像不像其他数字。于是，这些特征就形成了很多对整体图像的分析 and 判断。完成从图像数据到特征转换的过程就是线性部分要做的工作，这就是前面提到的“总结和判断”。而前面提到的参数  $W$  就是上面那些描述的数学化表示，也正是完成这个转换的关键。

这就是对线性部分功能的直观解释，听上去只要找到这些“描述”，问题就可以解决了。但是实际上这些“描述”全部以参数的形式表示，它并不已知，也不需要人为设定，它是模型要求解的内容。后面的章节会介绍求解其他参数的方法。

### 3.2 非线性部分

线性部分汇总之后的结果将被传递到非线性部分。非线性部分和线性部分相比，在形式上更灵活，它有一系列的“套路”函数可供选择，每个“套路”都有自己的特点。本节将介绍两个经典的非线性函数。首先介绍的是 Sigmoid 函数。它的函数形式如下所示：

$$f(x) = \frac{1}{1 + e^{-x}}$$

图像就是下面这条 S 型曲线，如图 3-4 所示。

这个函数的输入正是我们上一步线性部分的输出  $z$ ，此时  $z$  的取值范围在  $(-\infty, +\infty)$ ，经过这个函数就变成了  $(0,1)$ 。

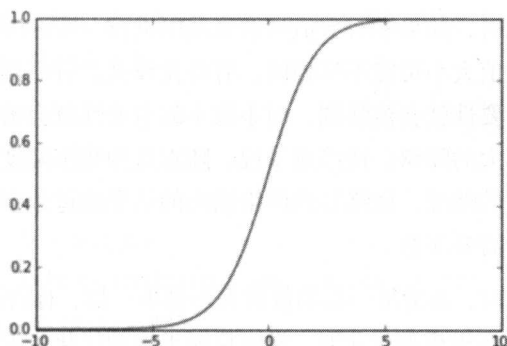


图 3-4 Sigmoid 曲线

另一个比较有名的非线性函数——双曲正切函数 Tanh，它的函数形式如下所示：

$$f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

它的图像如图 3-5 所示。

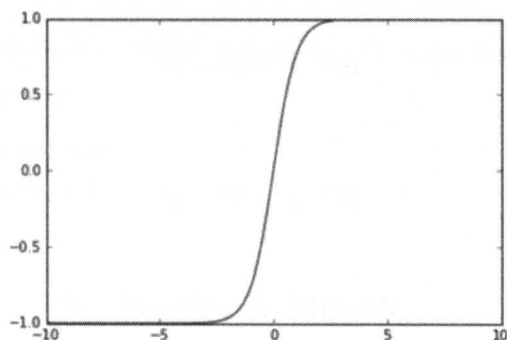


图 3-5 Tanh 函数

这个函数虽然形式长得很复杂，但看上去和前面的 Sigmoid 形状也差不多。可以看出，它的函数范围和前面的 Sigmoid 不同，是  $(-1, 1)$ 。

看完了上面这两个蛇形函数，读者也许会问：为什么模型需要非线性部分做这个函数转换呢，它有什么作用呢？

非线性部分在模型中有很多作用，其中一个作用就是对数据的归一化。不管前面的线性部分做了怎样的工作，到了非线性这里，所有的数值将被限制在某个范围内，比方说 Sigmoid 函数，它会将数据限制在  $(0, 1)$  的范围中。这样后面的网络层如果要基于前面网络层的数据继续计算，网络内部的数值的范围就相对可控了。不然的话，就有可能造成两个麻烦。

1. 从正向计算网络时，如果不对数值的取值范围进行一定的限制，那么在下一层网络中，输入的数值大小可能不尽相同，有些比较大，有些比较小，那么在计算中那些大数字的重要性就会被强调，而小数字的重要性就会被忽略。这对下面的计算来说会造成很大的障碍：往严重了说，随着这种数字幅度不断扩大，有可能遇到数值爆炸溢出的情况，最终导致网络输出的结果超过数字能表示的范围。这一点在深层网络中需要注意。
2. 从反向计算网络时，如果每一层的数值大小都不一样，有的范围在  $(0, 1)$ ，有的在  $(0, 10000)$ ，那么在做模型优化时，设定反向求导的优化步长就会充满挑战——如果设置过大，那么梯度较大的维度就会因为过量更新而造成无法预期的结果；如果设置过小，那么梯度较小的维度得不到充分的更新就难以有提升（这一部分将在后面章节详细介绍）。

非线性部分的另外一个作用是打破之前的线性映射关系。如果全连接层没有非线性部分，只有线性部分，那么在模型中叠加多层神经网络是没有意义的，因为多层神经网络可以直接退化成一层神经网络。

这里假设有一个两层全连接神经网络，其中没有非线性层，那么对于第一层有：

$$W^0 \cdot x^0 + b^0 = z^1$$

对于第二层有：

$$W^1 \cdot z^1 + b^1 = z^2$$

两式合并，有：

$$\begin{aligned} W^1 \cdot (W^0 \cdot x^0 + b^0) + b^1 &= z^2 \\ W^1 \cdot W^0 \cdot x^0 + (W^1 \cdot b^0 + b^1) &= z^2 \end{aligned}$$

所以我们只要令  $W^{0'} = W^1 \cdot W^0$ ， $b^{0'} = W^1 \cdot b^0 + b^1$ ，就可以用一层神经网络表示之前的两层神经网络了。所以从另一方面来说，非线性层的加入，使得多层神经网络的存在有了意义。

### 3.3 神经网络的模样

本节就要把全连接层的两个部分结合起来，看看它到底长成什么模样。由于神经网络充满了复杂性与多变性，因此接下来本节会采用递进的方式展示它的图像：第一步先设计一个简单的神经网络并将其展示出来，然后不断地增加它的复杂性，看它的图

像能变成什么样子。下面我们首先给出全连接层的代码，其中非线性部分采用 Sigmoid 函数，这段代码将贯穿下面的实验。

```
class FC:
    def init(self, in_num, out_num, lr = 0.01):
        self._in_num = in_num
        self._out_num = out_num
        self.w = np.random.randn(out_num, in_num) * 10
        self.b = np.zeros(out_num)
    def _sigmoid(self, in_data):
        return 1 / (1 + np.exp(-in_data))
    def forward(self, in_data):
        return self._sigmoid(np.dot(self.w, in_data) + self.b)
```

代码的内容并不多，这里就不去详细介绍了。值得一提的是，构造函数中模型会对参数中的  $w$  进行随机初始化，也就是说会随机一个神经网络出来。为了方便可视化，这里将输入的维度设置为 2，输出的维度设置为 1，这样的函数图像就可以被直接可视化。

首先隆重请出 1 号选手——只有一层全连接层的神经网络。实际上，对于只有一层且只有一个输出的神经网络，它的形式和逻辑回归（Logistic Regression）模型的形式相同。相关的代码如下所示：

```
x = np.linspace(-10,10,100)
y = np.linspace(-10,10,100)
X, Y = np.meshgrid(x,y)
X_f = X.flatten()
Y_f = Y.flatten()
data = zip(X_f, Y_f)

fc = FC(2, 1)
Z1 = np.array([fc.forward(d) for d in data])
Z1 = Z1.reshape((100,100))
draw3D(X, Y, Z1)
```

生成的结果图像如图 3-6 所示。

这个函数的样子和一个标准的 Logistic Regression 模型函数完全一样，长得像一个台阶。经过多次随机测试，基本上它都是这个形状，只不过随着权重的数值不断变化，这个“台阶”会旋转到不同的方向，但归根结底还是一个台阶。



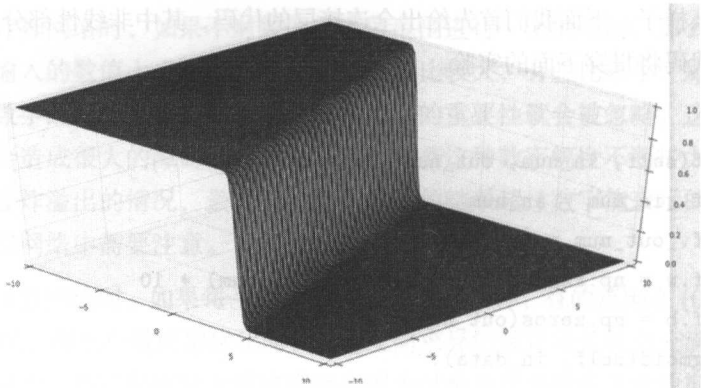


图 3-6 1 层神经网络图像

这也说明一层神经网络和我们想要的神经网络差距有点大，这个模型本质上只拥有线性分类器的实力，随着神经网络的层数不断增加，神经网络模型比逻辑斯特回归模型要复杂得多。那么，它复杂的样子是什么样呢？我们给神经网络再加一层，代码如下所示：

```
fc = FC(2, 3)
fc.w = np.array([[0.4, 0.6],[0.3,0.7],[0.2,0.8]])
fc.b = np.array([0.5,0.5,0.5])

fc2 = FC(3, 1)
fc2.w = np.array([0.3, 0.2, 0.1])
fc2.b = np.array([0.5])

Z1 = np.array([fc.forward(d) for d in data])
Z2 = np.array([fc2.forward(d) for d in Z1])
Z2 = Z2.reshape((100,100))

draw3D(X, Y, Z2)
```

这次暂时不用随机参数，而是设置了几个固定数值来看看效果。从上面的代码可以看出，参数设置得很用心。两层的参数全都是正数，它的图像如图 3-7 所示。

看上去比之前的台阶“柔软”了一些，但归根结底还是很像一个台阶，完全感受不到神经网络的强大。那就给神经网络加点负参数，看看会不会出现奇迹。

```
fc = FC(2, 3)
fc.w = np.array([[ -0.4, 1.6],[ -0.3,0.7],[0.2,-0.8]])
```

```

fc.b = np.array([-0.5,0.5,0.5])

fc2 = FC(3, 1)
fc2.w = np.array([-3, 2, -1])
fc2.b = np.array([0.5])

Z1 = np.array([fc.forward(d) for d in data])
Z2 = np.array([fc2.forward(d) for d in Z1])
Z2 = Z2.reshape((100,100))

draw3D(X, Y, Z2)

```

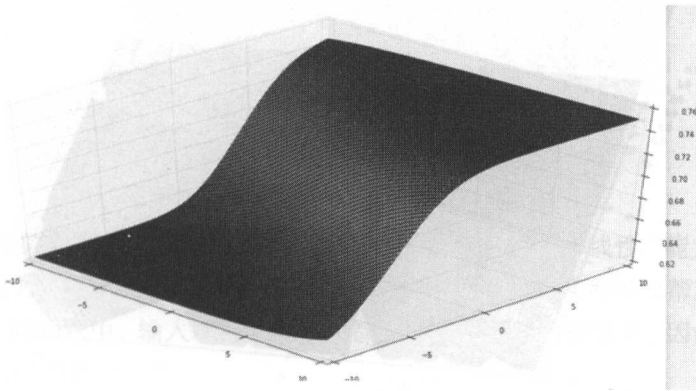


图 3-7 2层神经网络图像

可以看出部分参数已经被设置成负数了，它的图像如图 3-8 所示。

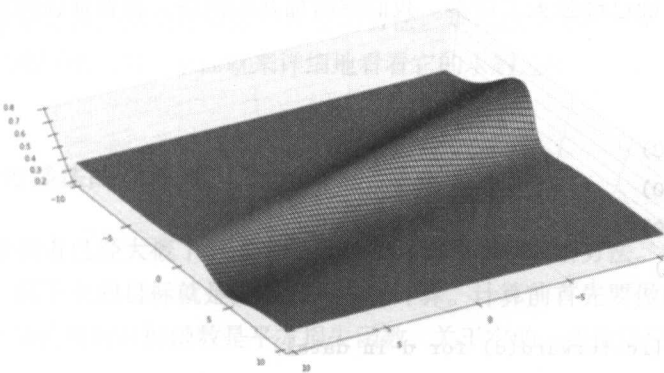


图 3-8 随机的 2 层神经网络图像

加了负参数后，模型终于看上去不那么像台阶了，两层神经网络的非线性能力开始显现。看完了两个相对简单的图像，下面就把参数交给随机“大帝”，看看它能带来什么惊喜：

```
fc = FC(2, 100)
fc2 = FC(100, 1)

Z1 = np.array([fc.forward(d) for d in data])
Z2 = np.array([fc2.forward(d) for d in Z1])
Z2 = Z2.reshape((100,100))
draw3D(X, Y, Z2,(75,80))
```

随机得到的一张图像如图 3-9 所示。

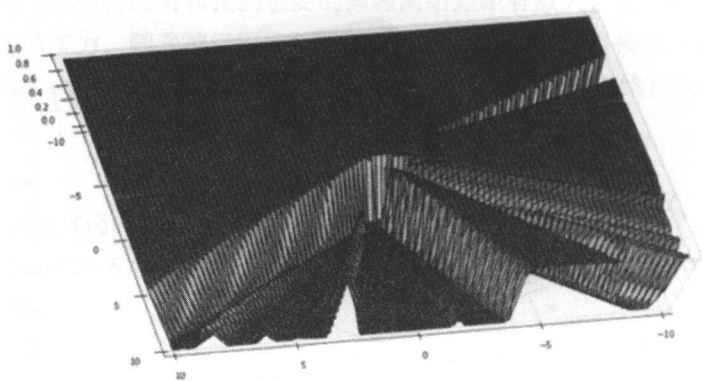


图 3-9 更复杂的随机 2 层神经网络图像

这时候模型的非线性特点已经非常明显了，这个模型和最初的“台阶”模型相比，在非线性表达方面已经强大了许多。我们继续加几层网络，看看神经网络还能变成什么样。

```
fc = FC(2, 10)
fc2 = FC(10, 20)
fc3 = FC(20, 40)
fc4 = FC(40, 80)
fc5 = FC(80, 1)

Z1 = np.array([fc.forward(d) for d in data])
Z2 = np.array([fc2.forward(d) for d in Z1])
Z3 = np.array([fc3.forward(d) for d in Z2])
```

```

Z4 = np.array([fc4.forward(d) for d in Z3])
Z5 = np.array([fc5.forward(d) for d in Z4])
Z5 = Z5.reshape((100,100))
draw3D(X, Y, Z5,(75,80))

```

图 3-10 又复杂了许多，这个形状已经很难用语言描述了。

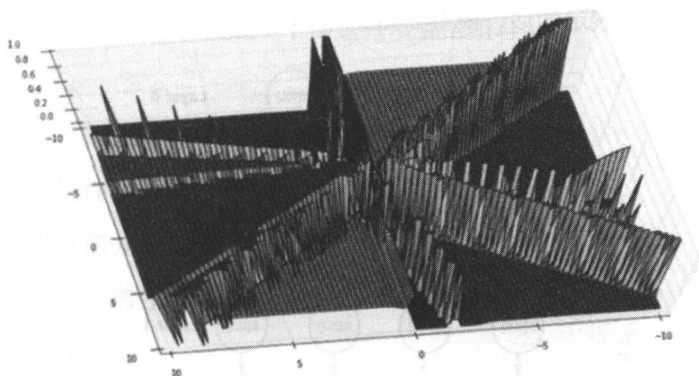


图 3-10 更复杂的多层随机网络图像

从上面的实验中可以看出，层数越高，模型所能表现出的非线性“能力”越强，“脑洞”开得也越大。如果遇到的问题像上面这样复杂，那么这个模型可以帮助我们解决问题。当然在实际问题中，输入和输出的维度都有可能比上面的模型多，因此模型真正的复杂度恐怕难以想象。

## 3.4 反向传播法

反向传播法是神经网络计算模型梯度的方法。利用这个方法，多层神经网络的梯度可以被很好地解耦求出。下面就来详细地看看它的求解过程。

### 3.4.1 反向传播法的计算方法

在第 2 章读者已经大概了解了当代机器学习的建模和求解方法。在了解了全连接层的形式后，接下来的目标就是完成它的求解计算。计算前首先要做的事情就是定义目标函数。本节使用的目标函数是平方损失函数，关于它的一些性质已经在 2.8 节中介绍过：

$$\text{Loss}(y, t) = \frac{1}{2}(y - t)^2$$

为了更好地描述反向传播的算法，这里要定义一个十分简单的双层神经网络，它的结构如下所示。

- 1. 输入的数据是 2 维。
- 2. 第一层神经网络的输入也是 2 维，输出是 4 维，非线性部分采用 Sigmoid 函数。
- 3. 第二层神经网络的输入也是 4 维，输出是 1 维，非线性部分采用 Sigmoid 函数。

如果用图的形式表示的话，模型如图 3-11 所示。

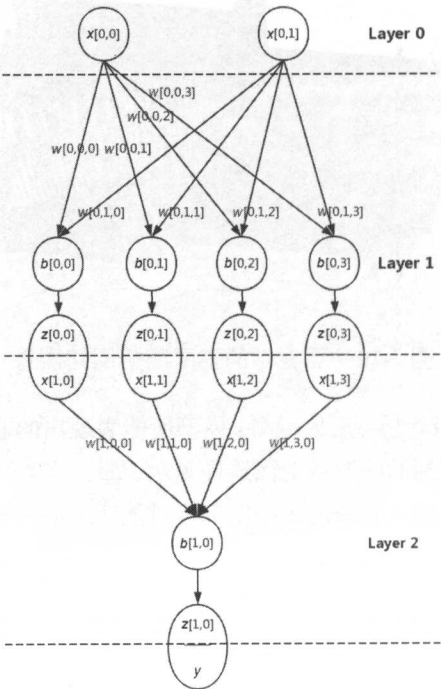


图 3-11 神经网络样例图示

第 2 章提到了函数优化的基本流程，那就是先为参数选定某个初始值，然后用迭代优化的方式寻找最优的参数值。迭代优化的关键就是求解参数关于目标函数的偏导数，下面的内容主要就是推导神经网络中参数的偏导数。推导公式本身不需要太多的数学知识，只需要有耐心。推导的过程要步步递进：首先解决单个数据的推导过程，然后将整个计算过程解耦合，最后把计算扩展到一批（batch）数据上。

首先要做的是列出所有待求的偏导数。

网络第一层：

$$\frac{\partial \text{Loss}}{\partial w_{00}^0}, \frac{\partial \text{Loss}}{\partial w_{01}^0}, \frac{\partial \text{Loss}}{\partial w_{02}^0}, \frac{\partial \text{Loss}}{\partial w_{03}^0}, \frac{\partial \text{Loss}}{\partial w_{10}^0}, \frac{\partial \text{Loss}}{\partial w_{11}^0}, \frac{\partial \text{Loss}}{\partial w_{12}^0}, \frac{\partial \text{Loss}}{\partial w_{13}^0}$$

网络第二层：

$$\frac{\partial \text{Loss}}{\partial w_{00}^1}, \frac{\partial \text{Loss}}{\partial w_{10}^1}, \frac{\partial \text{Loss}}{\partial w_{20}^1}, \frac{\partial \text{Loss}}{\partial w_{30}^1}, \frac{\partial \text{Loss}}{\partial b_0^1}$$

只要求出上面列出的 13 个偏导数，后面就可以用它的相反数乘以步长进行优化迭代。说实话，直接求解这些梯度确实有点难，因为计算公式还是有一些复杂。这时，微分世界的一大神器——链式求导就来了。如果把神经网络的计算过程拆解成一个个小的部分，那么它的计算过程就变成了下面这些步骤的组合。

1. 输入数据  $x^0$ 。
2. 第一层的线性部分输出  $z^0$ 。
3. 第一层的非线性部分输出  $x^1$ 。
4. 第二层的线性部分输出  $z^1$ 。
5. 第二层的非线性部分输出  $y$ 。
6. 二次损失函数 Loss。

求导的过程需要将上面列出的步骤反向进行。首先是第二层网络的计算公式反推：

$$\begin{aligned}\text{Loss} &= \frac{1}{2}(y - t)^2 \Rightarrow \frac{\partial \text{Loss}}{\partial y} = y - t \\ y &= \frac{1}{1 + e^{-z^1}} \Rightarrow \partial y / \partial z^1 = y(1 - y) \\ z_0^1 &= \sum_{i=0}^3 w_{i0}^1 x_i^1 + b_0^1 \Rightarrow \frac{\partial z_0^1}{\partial w_{00}^1} = x_0^1 \\ z_0^1 &= \sum_{i=0}^3 w_{i0}^1 x_i^1 + b_0^1 \Rightarrow \frac{\partial z_0^1}{\partial b_0^1} = 1\end{aligned}$$

到这里，我们已经顺利求出第二层的所有参数的导数了。下面是第一层网络的计算公式反推：

$$\begin{aligned}z_0^1 &= \sum_{i=0}^3 w_{i0}^1 x_i^1 + b_0^1 \Rightarrow \frac{\partial z_0^1}{\partial x_0^1} = w_{00}^1 \quad (\text{同层的其他参数类似}) \\ x_0^1 &= \frac{1}{1 + e^{-z_0^0}} \Rightarrow \frac{\partial x_0^1}{\partial z_0^0} = x_0^1(1 - x_0^1) \quad (\text{同层的其他参数类似}) \\ z_2^0 &= \sum_{i=0}^1 w_{i2}^0 x_i^0 + b_2^0 \Rightarrow \frac{\partial z_2^0}{\partial w_{12}^0} = x_1^0 \quad (\text{同层的其他参数类似}) \\ z_0^0 &= \sum_{i=0}^1 w_{i0}^0 x_i^0 + b_0^0 \Rightarrow \frac{\partial z_0^0}{\partial b_0^0} = 1 \quad (\text{同层的其他参数类似})\end{aligned}$$

到这里，基本的运算准备已经完成，后面的事情就是把这些计算出来的小部分组合起来，比方说：

$$\frac{\partial \text{Loss}}{\partial w_{12}^0} = \frac{\partial \text{Loss}}{\partial y} \cdot \frac{\partial y}{\partial z^1} \cdot \frac{\partial z^1}{\partial x_2^1} \cdot \frac{\partial x_2^1}{\partial z_2^0} \cdot \frac{\partial z_2^0}{\partial w_{12}^0}$$

看着有点复杂是吧？可是实际上这里面大部分内容都已经在前面的步骤中计算好了，这里只需要把数据全部代入就可以。当然，如果严格按照链式法则进行推导计算，求导的公式会比这个更复杂，但是由于中间部分的一些偏微分项实际上等于 0 可以略去，因此看上去会简单一些。

而且，随着模型从高层网络向低层反向计算，那些已经计算好的中间结果也可以用于计算低层参数的梯度。所以经过整理，全部的计算过程可以如下表示。

1.  $\frac{\partial \text{Loss}}{\partial y} = y - t$
2.  $\frac{\partial \text{Loss}}{\partial z_0^1} = \frac{\partial \text{Loss}}{\partial y} \cdot y \cdot (1 - y)$
3.  $\frac{\partial \text{Loss}}{\partial w_{00}^1} = \frac{\partial \text{Loss}}{\partial z_0^1} \cdot x_0^1$ （同层的其他参数类似）
4.  $\frac{\partial \text{Loss}}{\partial x_0^1} = \frac{\partial \text{Loss}}{\partial z_0^1} \cdot w_0^1$ （同层的其他参数类似）
5.  $\frac{\partial \text{Loss}}{\partial b_0^1} = \frac{\partial \text{Loss}}{\partial z_0^1} \cdot 1$ （同层的其他参数类似）
6.  $\frac{\partial \text{Loss}}{\partial z_0^0} = \frac{\partial \text{Loss}}{\partial x_0^1} \cdot x_0^1 \cdot (1 - x_0^1)$ （同层的其他参数类似）
7.  $\frac{\partial \text{Loss}}{\partial w_{00}^0} = \frac{\partial \text{Loss}}{\partial z_0^0} \cdot x_0^0$ （同层的其他参数类似）
8.  $\frac{\partial \text{Loss}}{\partial b_0^0} = \frac{\partial \text{Loss}}{\partial z_0^0} \cdot 1$ （同层的其他参数类似）

以上就是计算的全过程了。经过了这个推演，参数的导数确实得到了。求解过程虽然有些烦琐，但是非常有逻辑，所以只要记住链式求导这个思路，再复杂的网络结构也可以这样步步为营地求出来。这样也就完成了单一数据的梯度求解。

### 3.4.2 反向传播法在计算上的抽象

虽然 3.4.1 节的方法可以计算出每一个参数的偏导数，但是每一次都这样计算实在有些烦琐。这意味着对于不同深度的网络，我们需要开发不同的求导计算逻辑。实际上神经网络的反向传播是有规律可循的，回过头看 3.4.1 节结尾的 8 个步骤，就会发现这 8 个步骤可以分成 3 个部分。

- 第 1 步完成模型输出值的梯度计算。
- 第 2~5 步完成了第 2 层神经网络的梯度计算。
- 第 6~8 步完成了第 1 层神经网络的梯度计算。

如果从更具体的角度来看每一层神经网络反向计算的内容，就会发现它们都完成了下面的梯度计算。

1. Loss 对本层线性部分输出  $z$  的梯度。
2. Loss 对本层线性部分  $w$  的梯度。
3. Loss 对本层线性部分  $b$  的梯度。
4. Loss 对本层线性部分输入  $x$  的梯度（它是前一层网络的输出）。

了解了这个模式，神经网络梯度计算模块化这件事情就变得容易了许多。在前向计算时，每一层使用同样的计算流程产生输出，并传递给后一层作为输入；同样，在后向计算时，每一层也使用同样的计算流程——计算上面的四个值，把梯度反向传给前一层输出。这样，每一个全连接层之间的运算变得相对独立，代码实现了很好的复用性。

### 3.4.3 反向传播法在批量数据上的推广

在实际模型训练过程中，每次只计算一个训练数据的梯度是不太可能的，不但因为这种训练方法浪费训练资源，而且单一数据计算产生的梯度很可能不准确。因此针对一批数据集中计算便成了必需，那么就需要用矩阵来解决问题。矩阵计算不是简单的维度扩展，因为矩阵乘法不满足交换律，所以相乘的顺序一定要弄清楚。如果顺序出了错误，最终结果就会有问题。

下面开始正式推导矩阵版的公式。这里还是用 3.4.2 节的那个例子——输入有 2 个元素，第一层有 4 个输出，第 2 层有 1 个输出。假设训练数据有  $N$  个，由此对所有相关的训练数据和参数做以下约定。

- 所有的训练数据按列存储，也就是说，如果把  $N$  个数据组成一个矩阵，那个矩阵的行等于数据特征的数目，矩阵的列等于  $N$ 。
- 线性部分的权值  $w$  由一个矩阵构成，它的行数为该层的输入个数，列数为该层的输出个数。如果该层的输入为 2，输出为 4，那么这个权值  $w$  的矩阵就是一个  $2 \times 4$  的矩阵。
- 线性部分的权值  $b$  是一个行数等于输出个数，列数为 1 的矩阵。



基于上面的规则，3.4.1 节中例子的批量数据版就可以画成图 3-12 所示的结构，可以看出里面的数据  $x$ 、 $z$  和参数  $w$ 、 $b$  都符合我们刚才对数据组织的定义。

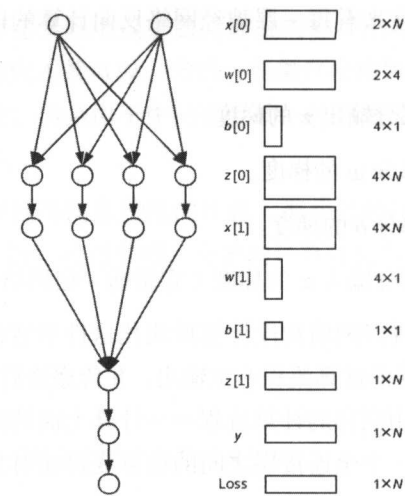


图 3-12 神经网络结构图

图 3-13 所示为批量数据版的全连接层前向计算，一共分为 5 步，其中前 4 步对应了两层网络的计算，最后一步用来计算 Loss。

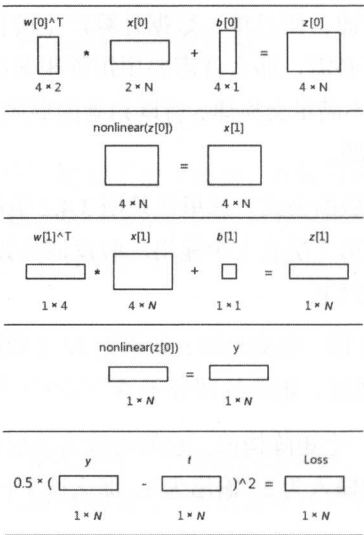


图 3-13 全连接层前向计算过程

图 3-14 所示为批量数据的全连接层反向计算全过程。由于具体的运算过程实际上和单一数据的方式类似，这里就不再赘述了。为了表达上的简洁，我们用符号  $g()$  表达 Loss 对指定变量的偏导数。同时为了更简洁地表达梯度计算的过程，在这个过程中我们对其中一个矩阵做了矩阵转置，这样可以确保最终输出维度的正确。

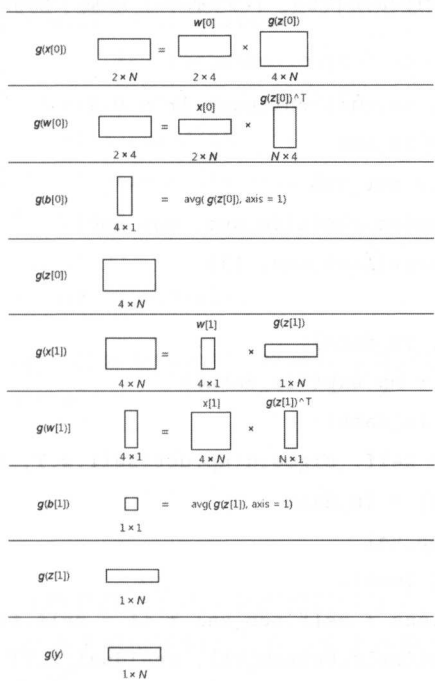


图 3-14 批量数据的全连接层反向计算全过程

希望读者能够认真地多看几遍图 3-14，最好能仔细地推导一遍，这样才能更好地掌握这个推导过程，尤其是理解为了维度对应做的矩阵转置。

看懂了上面这些图，接下来要做的就是对上面的内容进行总结，写出具有反向传播功能的全连接层代码。首先是目标函数的代码，这里的约定和上面相同， $y$  和  $t$  都是按列存储的，每一列都是一个训练样本。

```
class SquareLoss:
    def forward(self, y, t):
        self.loss = y - t
        return np.sum(self.loss * self.loss) / self.loss.shape[1] / 2
    def backward(self):
        return self.loss
```

为了代码的简洁，在前向运算时一些后向计算的信息都被预先保存起来，这样做的好处是在后向计算时能够简单点，坏处是这个类就不能具备多线程的特性了。由于这个代码主要用于演示，和真正工业级的代码并不完全相同。后面的全连接层也会采用同样的思路——前向计算时为后向计算准备运算数据，从而减少运算量。

接下来是矩阵版的全连接层代码，代码基本上是基于前面章节进行的扩展。

```
class FC:
    def __init__(self, in_num, out_num, lr = 0.1):
        self._in_num = in_num
        self._out_num = out_num
        self.w = np.random.randn(in_num, out_num)
        self.b = np.zeros((out_num, 1))
        self.lr = lr

    def _sigmoid(self, in_data):
        return 1 / (1 + np.exp(-in_data))

    def forward(self, in_data):
        self.top_val = self._sigmoid(np.dot(self.w.T, in_data) + self.b)
        self.bottom_val = in_data
        return self.top_val

    def backward(self, loss):
        residual_z = loss * self.top_val * (1 - self.top_val)
        grad_w = np.dot(self.bottom_val, residual_z.T)
        grad_b = np.sum(residual_z)
        self.w -= self.lr * grad_w
        self.b -= self.lr * grad_b
        residual_x = np.dot(self.w, residual_z)
        return residual_x
```

现在有了目标函数类和全连接类，还需要一个类把上面这两部分串联起来，也就是下面的 Net 类。为了后面的内容演示，这里在 Net 类初始化时对网络做了一些设定。

```
class Net:
    def __init__(self, input_num=2, hidden_num=4, out_num=1, lr=0.1):
        self.fc1 = FC(input_num, hidden_num, lr)
        self.fc2 = FC(hidden_num, out_num, lr)
        self.loss = SquareLoss()

    def train(self, X, y): # X are arranged by col
        for i in range(10000):
```

```

# forward step
layer1out = self.fc1.forward(X)
layer2out = self.fc2.forward(layer1out)
loss = self.loss.forward(layer2out, y)

# backward step
layer2loss = self.loss.backward()
layer1loss = self.fc2.backward(layer2loss)
saliency = self.fc1.backward(layer1loss)

layer1out = self.fc1.forward(X)
layer2out = self.fc2.forward(layer1out)
print 'X={0}'.format(X)
print 't={0}'.format(y)
print 'y={0}'.format(layer2out)

```

到此全连接层代码的编写就基本完成了，虽然优化过程还可以加入更多内容（例如正则化），但目前先写这么多。

### 3.4.4 具体的例子

下面用一个经典又十分简单的例子展示全连接层的效果，这个例子就是逻辑运算。本节将使用一个两层神经网络来模拟逻辑与运算。关于逻辑与运算相信大家已经十分了解了，下面是具体的代码：

```

# and operation result
X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]]).T
y = np.array([[0], [0], [0], [1]]).T

net = Net(2,4,1,0.1)
net.train(X,y)

```

以下是调用上面代码给出的结果，可以看出最终的结果效果还不错，经过 10000 轮的迭代，最终模型给出的结果和期望的结果十分相近。如果继续进行迭代，这个模型的精度还可以进一步提高，Loss 可以进一步减少，不过减少得会比较有限。

```

iter = 0, loss =0.105
=== Label vs Prediction ===
t=[[0 0 0 1]]
y=[[ 0.409  0.461  0.369  0.429 ]]

```

```
iter = 1000, loss =0.023
=== Label vs Prediction ===
t=[[0 0 0 1]]
y=[[ 0.044  0.226  0.177  0.686]]
iter = 2000, loss =0.006
=== Label vs Prediction ===
t=[[0 0 0 1]]
y=[[ 0.010  0.113  0.110  0.834]]
iter = 3000, loss =0.003
=== Label vs Prediction ===
t=[[0 0 0 1]]
y=[[ 0.005  0.078  0.078  0.884]]
iter = 4000, loss =0.002
=== Label vs Prediction ===
t=[[0 0 0 1]]
y=[[ 0.003  0.061  0.062  0.908]]
iter = 5000, loss =0.001
=== Label vs Prediction ===
t=[[0 0 0 1]]
y=[[ 0.002  0.051  0.052  0.923]]
iter = 6000, loss =0.001
=== Label vs Prediction ===
t=[[0 0 0 1]]
y=[[ 0.002  0.045  0.045  0.932]]
iter = 7000, loss =0.0008
=== Label vs Prediction ===
t=[[0 0 0 1]]
y=[[ 0.001  0.040  0.0402  0.939 ]]
iter = 8000, loss =0.0007
=== Label vs Prediction ===
t=[[0 0 0 1]]
y=[[ 0.001  0.037  0.037  0.945]]
iter = 9000, loss =0.000609513241467
=== Label vs Prediction ===
t=[[0 0 0 1]]
y=[[ 0.001  0.034  0.034  0.949]]
=== Final ===
```

```
X=[[0 0 1 1]
   [0 1 0 1]]
t=[[0 0 0 1]]
y=[[ 0.001  0.031  0.031  0.953]]
```

这个例子是万千例子中比较简单的一个，对于神经网络来说完成这个例子只能算是小儿科，后面将会有更复杂的例子。

### 3.5 参数初始化

下面来看看神经网络中的一个小陷阱。这个陷阱虽然一般不会被遇到，但是了解它还是十分有意义的。一直以来，参数初始化都是神经网络中一件十分重要的事情，一个实验可以说明一切问题。下面一起来看看如果把所有的参数初始化为 0，模型会有什么样的奇怪表现。代码如下：

```
X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]]).T
y = np.array([[0], [0], [0], [1]]).T

net = Net(2,4,1,0.1)
net.fc1.w.fill(0)
net.fc2.w.fill(0)
net.train(X,y)
print "=== w1 ==="
print net.fc1.w
print "=== w2 ==="
print net.fc2.w
```

直接看运行结果：

```
=== Final ===
X=[[0 0 1 1]
   [0 1 0 1]]
t=[[0 0 0 1]]
y=[[ 3.224e-04  2.223e-02  2.223e-02  9.577e-01]]
=== w1 ===
[[-2.490 -2.490 -2.490 -2.490]
 [-2.490 -2.490 -2.490 -2.490]]
=== w2 ===
```

```

[[-3.373]
 [-3.373]
 [-3.373]
 [-3.373]]

```

从训练结果看，模型的预测结果和理想结果差距也不算大，但令人惊讶的是训练完成后每一层的参数的数值都是完全相同的，经过 10000 轮的训练，它们之间竟然没有产生不同。从这个实验可以看出，如果把模型的参数初始化成 0，训练后模型实际上会有退化的现象发生，所有的参数完全一样，这样起不到多个“角度”分析的效果。

那么这个现象是不是只有将参数初始化为 0 才会发生呢？如果把参数初始化为不为 0 的某个值呢？

```

X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]]).T
y = np.array([[0],[0],[0],[1]]).T

```

```

net = Net(2,4,1,0.1)
net.fc1.w.fill(1)
net.fc2.w.fill(0)
net.train(X,y)
print "=== w1 ==="
print net.fc1.w
print "=== w2 ==="
print net.fc2.w

```

结果如下所示。

```

=== Final ===
X=[[0 0 1 1]
 [0 1 0 1]]
t=[[0 0 0 1]]
y=[[ 0.004  0.028  0.028  0.969]]
=== w1 ===
[[ 2.482  2.482  2.482  2.482]
 [ 2.482  2.482  2.482  2.482]]
=== w2 ===
[[ 3.231]
 [ 3.231]
 [ 3.231]
 [ 3.231]]

```

虽然训练结果不错，但是每层的参数依然完全相同。看来这和初始化为 0 没有关系，只要初始化时把所有参数设置成相同的值，那么参数之间就无法训练出不同的数值。这是为什么呢？其实想要解释这个问题，不如从这个例子入手，把上面的例子做一遍完整的推演，去了解模型内部的计算过程，将优化第一轮迭代的中间结果完整地输出出来，就可以看出其中的奥秘。首先是第一层前向计算的结果 `top_val`：

```
top_val=
[[ 0.5  0.731  0.731  0.880]
 [ 0.5  0.731  0.731  0.880]
 [ 0.5  0.731  0.731  0.880]
 [ 0.5  0.731  0.731  0.880]]
```

由于参数完全一样，所以每一组数据计算出来的几种“观察角度”观察出来的结果是一样的，每一组数据输出的每一维都是一样的。

接下来是第二层的输出结果 `top_val`，也就是模型输出的结果 `y`：

```
top_val=
[[ 0.5  0.5  0.5  0.5]]
```

到这里前向计算就结束了，下面开始反向计算，首先是第二层的两个中间结果——`residual_z` 和 `grad_w`：

```
residual_z=
[[ 0.125  0.125  0.125 -0.125]]
grad_w=
[[ 0.135]
 [ 0.135]
 [ 0.135]
 [ 0.135]]
residual_x=
[[-0.001 -0.001 -0.001  0.001]
 [-0.001 -0.001 -0.001  0.001]
 [-0.001 -0.001 -0.001  0.001]
 [-0.001 -0.001 -0.001  0.001]]
```

可以看到，虽然 `residual_z`——也就是 Loss 对这一层线性输出的梯度不同，但是根据反向传播的公式，参数和输入的梯度又变得完全相同。其中 `grad_w` 是第二层 `top_val` 和 `residual_z.T` 相乘的结果，可以看出因为第一层的 `top_val` 每一行完全相同，所以它们与同一列相乘的结果也就完全相同，同理还有 `residual_x`。



那么，第一层的结果呢？

```
residual_z=
[[-0.0004 -0.0003 -0.0003  0.0001]
 [-0.0004 -0.0003 -0.0003  0.0001]
 [-0.0004 -0.0003 -0.0003  0.0001]
 [-0.0004 -0.0003 -0.0003  0.0001]]

grad_w=
[[-0.0001 -0.0001 -0.0001 -0.0001]
 [-0.0001 -0.0001 -0.0001 -0.0001]]

residual_x=
[[-0.0016 -0.0013 -0.0013  0.0007]
 [-0.0016 -0.0013 -0.0013  0.0007]]
```

也是清一色地相同。通过这一轮迭代的分析，相信读者已经发现其中的问题了，这里就不对后面的迭代做分析了。总体来说，相同的参数会导致相同的中间结果，同时会产生相同的参数更新量。对于神经网络来说，这相当于网络的退化——本来可以从很多角度分析汇总数据的，现在只能从一个角度分析。为了打破这样的相同，必须采取一些手段不让参数变得相同，不然模型退化的事情就会发生，例如随机初始化。好在现在的开源框架都为大家默认提供了随机生成初始参数的方法，也基本不会发生这样的事情了。

## 3.6 总结

本章完成了第一个核心概念——全连接层的介绍。作为神经网络的基础，其中的很多概念值得读者研究和思考。让我们来回顾一下。

- 线性部分：从不同角度分析汇总数据。
- 非线性部分：打破线性关系，限制数值范围。
- 神经网络的图像：网络越复杂，图像越复杂。
- 反向传播：链式法则解决导数求解。
- 模块计算与矩阵运算：神经网络的工程化实现。
- 初始化：参数的初始化对防止网络退化很重要。

# 4

## CNN 的基石：卷积层

本章的主要内容是介绍卷积神经网络的另一个核心模型层——卷积层。卷积操作是图像处理中很经典的一种操作，以它为中心的理论也十分丰富。本章首先介绍卷积操作的定义，然后探讨卷积层和全连接层之间的异同，接下来给出卷积层反向计算的方法，最后介绍在深度学习时代知名的非线性函数——ReLU。

### 4.1 卷积操作

#### 4.1.1 卷积是什么

从结构上看，卷积层的组成和全连接层类似，它也由线性部分和非线性部分组成。第3章介绍了全连接层的线性部分，这一部分主要完成了汇总计算的过程，汇总的对象是所有数据。卷积层的线性部分同样要完成汇总计算，不过它的汇总是局部数据的汇总，这种操作被称为卷积。

卷积操作是分析数学中的一种重要运算。它的原理比较抽象而且和本书的主题相关度不大，这里就不去详细介绍了，下面将重点关注离散图像下的卷积运算。卷积运算由图像数据和卷积核两部分数据合作完成，图像数据中的每一个和卷积核等大小的区域都会和卷积核完成元素级（Element-wise）乘法，并将乘法得到的结果加和汇聚成一个数字。由于每个区域都会得到这样的数字，所有的数字根据相对位置拼合起来，就是卷积计算最后的结果。

首先，卷积层的计算只会做某一部分的数据汇总计算，比方说在以某一像素为中心的某个区域内进行汇总计算，而具体计算公式和全连接层完全相同，都是像素值和权重的乘加操作，如图 4-1 所示。

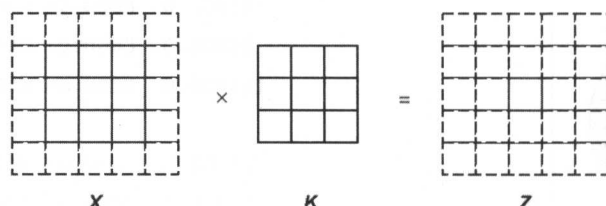


图 4-1 卷积运算操作示意

其次，卷积层在完成局部汇总计算后，会保持计算前的相对位置。例如，左右相邻的两个像素，以左边像素为中心的汇总结果还是会在以右边像素为中心的汇总结果的左边。这使得卷积层拥有一个与全连接层不同的特点，那就是卷积层的输入输出及中间结果都保持一定的空间形状。一般来说，图像的特征信息在卷积层中经常以 3 维的形式保存： $C \times H \times W$ （有些深度学习框架，例如 TensorFlow，也会采用  $H \times W \times C$  的形式表示）。其中  $C$  表示通道，每一个通道相当于图像中每个像素的一种特征。另外的  $H$  和  $W$  表示图像的高和宽。在第 3 章中已经介绍过，全连接层的输入输出一般是以 1 维向量的形式保存，而且向量中数值顺序互换并不影响其语义。从这里就而且可以看出全连接层和卷积层的一些差别。

卷积层中的局部汇总计算方法一般都是用“相关”这个操作来实现的，这个概念和本书的内容关联不大，这里就不展开介绍了。那么，相关和卷积的计算有什么差别呢？卷积在做汇总计算时，会首先把参数矩阵旋转 180 度，而相关操作就不用这样做，所以采用相关操作在进行前向计算时可以减少一次把卷积核转一圈的计算，这样在程序上线运行时能够更快些。实际上也可以假设这里就是要进行卷积操作，只不过是把参数全部预先旋转了 180 度，这样在做卷积操作时，就不用再转了。

了解了计算方法后，接下来看看“卷积层”操作的基本代码，为了代码的间接性，这里只展示了对单一数据的卷积前向计算，同时还省略了一些常用的参数配置：

```
import numpy as np
import matplotlib.pyplot as plt
def conv2(X, k):
    x_row, x_col = X.shape
    k_row, k_col = k.shape
    ret_row, ret_col = x_row - k_row + 1, x_col - k_col + 1
    ret = np.empty((ret_row, ret_col))
```

```

for y in range(ret_row):
    for x in range(ret_col):
        sub = X[y : y + k_row, x : x + k_col]
        ret[y,x] = np.sum(sub * k)
return ret

```

```

class ConvLayer:
    def __init__(self, in_channel, out_channel, kernel_size):
        self.w = np.random.randn(in_channel, out_channel, kernel_size,
                                   kernel_size)
        self.b = np.zeros((out_channel))
    def _relu(self, x):
        x[x < 0] = 0
        return x
    def forward(self, in_data):
        # assume the first index is channel index
        in_channel, in_row, in_col = in_data.shape
        out_channel, kernel_row, kernel_col = self.w.shape[1], self.w.
        shape[2], self.w.shape[3]
        self.top_val = np.zeros((out_channel, in_row - kernel_row + 1,
                                   in_col - kernel_col + 1))
        for j in range(out_channel):
            for i in range(in_channel):
                self.top_val[j] += conv2(in_data[i], self.w[i, j])
            self.top_val[j] += self.b[j]
        self.top_val[j] = self._relu(self.top_val[j])
        return self.top_val

```

这一堆复杂的代码恐怕够读者看上一会儿了，但愿这些代码没有吓跑大家。实际上，代码复杂的主要原因是数据的维度比较高，如果大家看懂了全连接层的计算，那么对这个代码稍做分析也一样能看明白。

前面的介绍主要涉及了最基本的卷积操作，实际上卷积层的计算比全连接层要复杂一些，它的内部还有一些可以调整的参数。一般来说，卷积层中有两个常用的参数——**stride** 和 **padding**。

stride 有点像一个循环函数的步进量，在上面介绍的算法中，当计算完一个像素后，下一个计算的像素通常是这个像素右边的一个像素（本行末尾除外），这相当于 stride=1 的场景。如果 stride=2，就表示每计算完一个像素后要跳过一个像素再进行计算，这样

最终参与计算的输入数据就会少很多。它实际上起到了采样的作用。stride 的效果如图 4-2 所示。

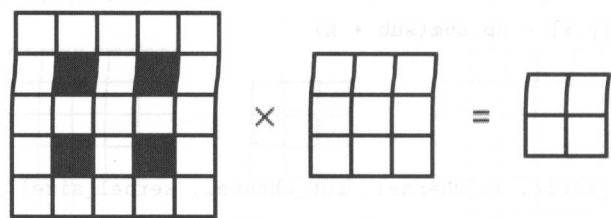


图 4-2 stride 计算示意图（图中深色的像素块为 stride=2 时参与计算的中心像素）

padding 是一种维持图像维度的方法，它的作用是“填边”。如果在一张图上直接做卷积操作，就会遇到一个尴尬的问题：对于那些处于边角的像素，由于它们的周围邻域大小小于卷积核，因此无法参与计算，但如果它们不能参与计算，实际上相当于计算结果的维度比输入数据小。这样有时也会造成困扰，因此为了解决这样的问题，图像周围可以填上一圈空的数据，这样每一个输入数据中的像素都可以参与卷积计算，图像的维度就不会缩小。

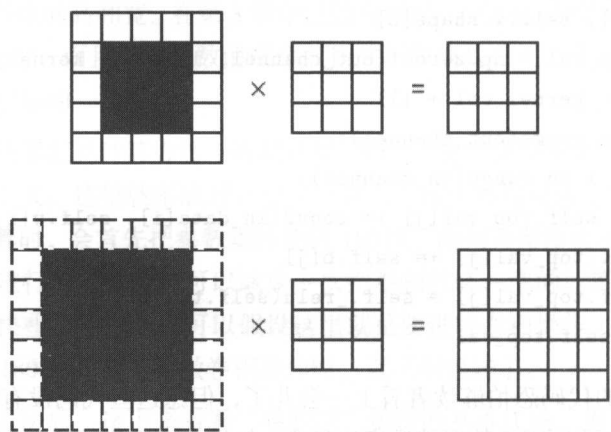


图 4-3 padding 计算示意图（图中展示了卷积核宽度为 3 时，padding=0 和 padding=1 情况下的输出维度）

padding 的效果如图 4-3 所示，stride 和 padding 会对输出的尺度产生影响。当 stride 和 padding 不产生作用时，它们的默认值分别是 stride=1，padding=0，此时输入和输出维度的关系公式如下所示：

$$H_{out} = H_{in} - H_{kernel} + 1$$

$$W_{out} = W_{in} - W_{kernel} + 1$$

当 stride 和 padding 产生作用时，公式如下所示：

$$H_{\text{out}} = \frac{H_{\text{in}} + 2H_{\text{padding}} - H_{\text{kernel}}}{H_{\text{stride}}} + 1$$

$$W_{\text{out}} = \frac{W_{\text{in}} + 2W_{\text{padding}} - W_{\text{kernel}}}{W_{\text{stride}}} + 1$$

很显然，当 stride=1，padding=0 时，两组公式是等价的。

### 4.1.2 卷积层效果展示

下面来介绍卷积层的可视化效果。在第 3 章中介绍了全连接层汇总，实际上卷积层的计算也是一个汇总的过程，那么这个汇总和全连接层的汇总有什么不同呢？卷积操作毕竟是图像处理中的经典操作之一，这个问题还是值得大书特书的。为了更清楚地介绍这些内容，这里以一张光学字符识别（Optical Character Recognition, OCR）的训练数据做例子：

```
import cv2
mat = cv2.imread('conv1.png',0)
row,col = mat.shape
in_data = mat.reshape(1,row,col)
in_data = in_data.astype(np.float) / 255
plt.imshow(in_data[0], cmap='Greys_r')
```

这段代码会显示出如图 4-4 所示的图像。

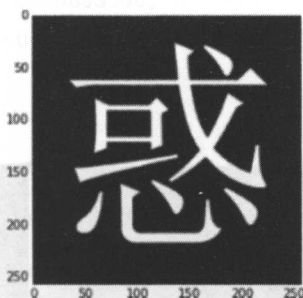


图 4-4 OpenCV 显示的示例图像

接下来的卷积操作就要在这样的单通道（channel 数为 1）的图片上进行。在图像处理领域，卷积操作可以看作是对图像做滤波操作。滤波算法有很多，下面一起来看看这些操作可视化的效果。

首先是均值滤波（Mean Filtering），这个滤波的特点是求出某个像素附近所有像素的平均值，并把这个平均值赋值给这个像素。它的代码如下所示：

```
meanConv = ConvLayer(1,1,5)
w = np.ones((5,5)) / (5 * 5)
meanConv.w[0,0] = w
mean_out = meanConv.forward(in_data)
plt.imshow(mean_out[0], cmap='Greys_r')
print w
```

```
[[ 0.04,  0.04,  0.04,  0.04,  0.04],
 [ 0.04,  0.04,  0.04,  0.04,  0.04],
 [ 0.04,  0.04,  0.04,  0.04,  0.04],
 [ 0.04,  0.04,  0.04,  0.04,  0.04],
 [ 0.04,  0.04,  0.04,  0.04,  0.04]]
```

结果如图 4-5 所示。

均值滤波在图像处理中可以起到模糊图像的作用，当然由于卷积核比较小，效果不是很明显。读者可以尝试把卷积核的维度设置得大些，再看看效果。

接下来是梯度计算滤波的代表——Sobel filter，这里定义一个计算纵向梯度的卷积核，如果一个像素点的纵向梯度非常大，那么这个点的结果会非常大。

```
sobelConv = ConvLayer(1,1,3)
sobelConv.w[0,0] = np.array([[ -1,-2,-1],[0,0,0],[1,2,1]])
sobel_out = sobelConv.forward(in_data)
plt.imshow(sobel_out[0], cmap='Greys_r')
```

图像结果如图 4-6 所示。

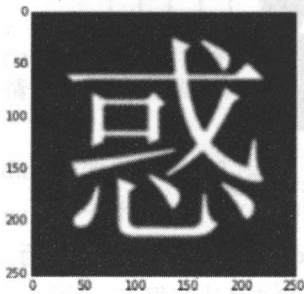


图 4-5 均值滤波结果图

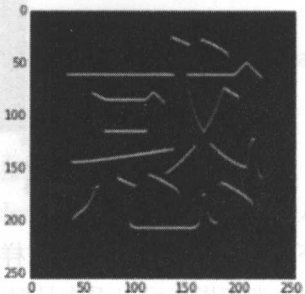


图 4-6 Sobel 滤波结果图

从结果看，文字中横向比划的上端被保留了下来。为什么呢？因为纵向梯度大的地方往往是下面一个像素值大，上面一个像素值小，这样每个比划的上端就被保存下来了。

最后是 Gabor filter，这也是一个经典的滤波算子。现在的一些深度学习的论文里都以自己模型的第一层能够学出类似 Gabor filter 的参数为荣，可见这个滤波器的影响力。下面来看看它的效果（以下 Gabor filter 的代码来自 wikipedia<sup>[1]</sup>）：

```
def gabor_fn(sigma, theta, Lambda, psi, gamma):
    sigma_x = sigma
    sigma_y = float(sigma) / gamma
    (y, x) = np.meshgrid(np.arange(-1,2), np.arange(-1,2))
    # Rotation
    x_theta = x * np.cos(theta) + y * np.sin(theta)
    y_theta = -x * np.sin(theta) + y * np.cos(theta)
    gb = np.exp(-.5 * (x_theta ** 2 / sigma_x ** 2 + y_theta ** 2 /
    sigma_y ** 2)) * np.cos(2 * np.pi / Lambda * x_theta + psi)
    return gb

print gabor_fn(2, 0, 0.3, 0, 2)
gaborConv = ConvLayer(1,1,3)
gaborConv.w[0,0] = gabor_fn(2, 0, 0.3, 1, 2)
gabor_out = gaborConv.forward(in_data)
plt.imshow(gabor_out[0], cmap='Greys_r')

[[-0.26763071 -0.44124845 -0.26763071]
 [ 0.60653066  1.          0.60653066]
 [-0.26763071 -0.44124845 -0.26763071]]
```

图像结果如图 4-7 所示。

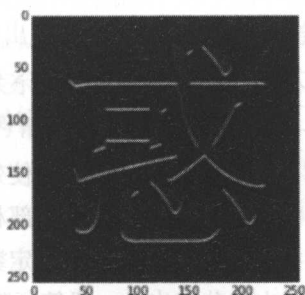


图 4-7 Gabor 滤波结果图



从结果可以看出，与 Sobel filter 类似，Gabor filter 同样可以起到边缘提取的作用。这次提取的似乎是汉字的笔画下端，这和它本身的功能描述是相符的。

从三种卷积操作的处理过程可以看出，不同卷积核的卷积操作确实对图像产生了不同的效果。在应用中，用于图像滤波的卷积核还有很多，但上面的三个卷积核已经具有代表性了，因此就不再一一展示了。

### 4.1.3 卷积层汇总了什么

解决了“是什么”这个问题，下面就来看看“为什么”这个问题。为什么要使用卷积层？与全连接层相比卷积层有什么优势？

卷积层的第一个优势是它的参数数量相对少一些。很显然，用全连接层代替卷积层是完全没有问题的，但是这样做的代价实在太大了。原始图像的维度相对而言比较大，如果采用全连接的话，参数数量将会有爆炸式的增长，这个计算量对于当代计算机来说同样是一个很大的挑战。试想一下，对于 MNIST 的数据集，如果模型的第一层是全连接层，它的输入是  $1 \times 28 \times 28$  的图像，输出是  $1 \times 1024$  的向量，那么它的参数已经达到 800 万个，而曾经的经典模型 LeNet 呢？它的网络结构如图 4-8 所示。

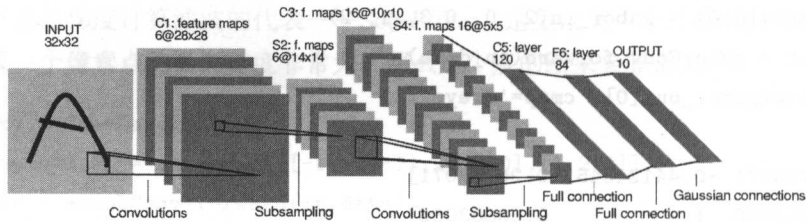


图 4-8 卷积神经网络 LeNet 架构图（图片来源：Lecun Y, Bottou L, Bengio Y, et al. Gradient-based learning applied to document recognition[J]. Proceedings of the IEEE, 1998, 86(11):2278-2324.）

LeNet 网络的第一层参数数量可以通过图 4-8 计算出来：输出的维度为  $6 \times 28 \times 28$ ，比上面提到的全连接层维度还大，但参数数量仅为： $1 \times 6 \times (5 \times 5 + 1) = 156$ ，两者的参数数量差距几万倍，而实际上两者的效果绝对不会有如此大的差距。

卷积层确实可以节省参数，但是节省参数会不会使效果变差呢？卷积层的第二个优势可以解释它的长处——卷积运算利用了图像的局部相关性。第 3 章介绍了全连接层的线性部分运算，它通过寻找整张图像的特征来确定数字内容，实际上除了这个粒度的相关性，图像的局部相关性也十分突出：一段笔画附近的像素往往是相同颜色的，一块背景区域的像素之间的颜色差距也不大。一个像素通常和周围的像素十分相近，除

非这个像素处于两个实体的边界，这就是最直接的局部相关。一个像素通常和距离很远的像素关系不大，因此卷积计算在关注了附近像素的同时，忽略了远处的像素，这也和实际图像表现出来的性质类似。

那么，如何消除这些局部相关性，使我们的特征变得少而精呢？卷积就是一种很好的方法。它只考虑附近一块区域的内容，分析这一小片区域的特点，这样针对小区域的算法可以很容易地分析出区域内的相关性。如果再加上 Pooling 层（可以理解为汇集、聚集的意思），从附近的卷积结果中再采样选择一些高价值的信息，丢弃一些重复低质量的信息，就可以做到对特征信息的进一步过滤处理，让特征向少而精的方向前进。

#### 4.1.4 卷积的另一种解释

4.1.3 节介绍了卷积操作解决图像局部相关性的思路，本节将从另一个角度分析卷积的功效。熟悉图像处理的读者一定都知道卷积定理，这个定理涉及图像处理的一大“黑科技”：傅立叶变换。

傅立叶变换可以被看作是对图像或者音频等数据的重新组织。它把数据从空间域的展示形式转变到频率域的形式。曾经有人这样比喻傅立叶变换：如果把看到的图像比作一道做好的菜，那么傅立叶变换就是找出这道菜具体的配料及各种配料的用量。这个方法的神奇之处在于不管这道菜是如何做的（按类别码放还是“大乱炖”），它都能将配料清晰地分出来。在图像处理中，配料的种类按频率进行划分。其中有些信息被称为低频信息，有些被称为高频信息。

低频信息一般被看作是整幅图像的基础，有点像一道菜中的主料。如果说这道菜是番茄炒蛋，那低频信息可以看作是番茄和蛋的大体形状和轮廓。高频信息一般是指那些变动比较大的，表达图像特点的信息，有点像一道菜的配料或者调料。对于番茄炒蛋，高频信息更像是番茄和蛋的纹理特征，菜中调料和点缀的材料。

卷积定理提到，两个矩阵的卷积的结果，等于两个矩阵在经过傅立叶变换后，进行元素级别的乘法（Element-wise Multiplication），再对结果进行反向傅立叶处理。如果用 FFT 表示傅立叶变换，IFFT 表示反向傅立叶变换，那么下面两个过程的结果是相同的（考虑到浮点数近似，可以认为相近就是相同）：

```
A = np.array()
B = np.array() # matrix
# method1
C = conv2(A,B)
# method2
```

```
FFT_A = FFT(A)
FFT_B = FFT(B)
C = IFFT(FFT_A * FFT_B)
```

真实的代码不会像上面这么简单，但是大体结构是相同的。看上去第二种方法比第一种方法啰嗦了不少，那么它的优势在哪里？回到前面展示卷积效果的例子中，下面来看看曾提到的那些滤波算法的卷积核经过傅立叶变换后的样子。

首先是均值滤波核，如图 4-9 所示。

然后是 Sobel 滤波核，如图 4-10 所示。

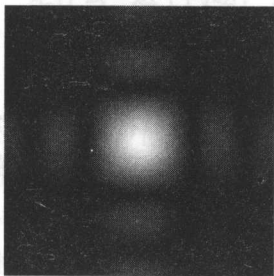


图 4-9 均值滤波核在傅立叶转换后的图像

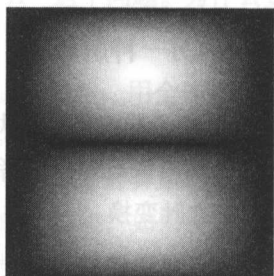


图 4-10 Sobel 滤波核在傅立叶转换后的图像

最后是 Gabor 滤波核，如图 4-11 所示。

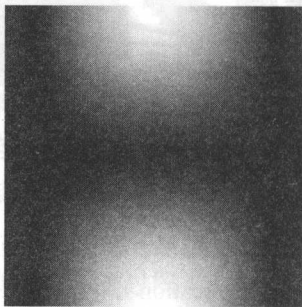


图 4-11 Gabor 滤波核在傅立叶转换后的图像

前面提到的傅立叶变换可以帮助我们分析出图像的高低频信息。在图 4-11 中，（经过 `fft_shift` 操作的）图像的正中心表示图像低频信息的强度（`magnitude`），越靠近中心的位置频率越低，越靠近边缘的位置频率越高。由于最终要进行元素级的乘法，如果卷积核在某个频率的数值比较低，经过乘法后的输入数据在这个频率的数据也会变小。滤波核在某个频率的数值为 0，说明卷积算法计算后会舍弃掉这部分信息。

如果我们将上面的图片想象成以中心为原点的直角坐标系图，就可以看出：

- 均值滤波核会保留中心附近和坐标轴附近的信息。
- Sobel 滤波核会保留  $y$  轴上下的信息，丢弃中间的信息和  $x$  轴两边的信息。
- Gabor 滤波核和 Sobel 滤波核类似，但是保留的内容会更少，更倾向于保留远离中心的像素。

从这个角度来分析结果，Sobel 滤波核和 Gabor 滤波核更倾向于保留高频信息而弱化低频信息，而均值滤波核则是弱化高频信息而保留低频信息，所以均值滤波和另外两个滤波算法的作用不同。这也和我们观察到的结果一致。前面提到一些文章宣称自己的模型第一层的参数像 Gabor filter，也说明他们提到的模型卷积核的作用是保留高频牺牲低频。

那么从这个角度分析，卷积层的意义在哪里？如果现在的任务是给出一盘番茄炒蛋，问这盘菜是哪位师傅做的（记得小时候在饭店吃饭时，菜盘边上经常会有一张小纸条，上面写着“ $\times\times$  号厨师为您服务”，现在不常见了），那么盘子里的数量众多的番茄和鸡蛋不一定能帮助我们找到厨师，而其中佐料的分量更有助于我们找出厨师的做菜风格。当然，这里我们假设厨师有自己的风格，且正常发挥没有失误。如果我们想知道这是一道什么菜，低频信息的代表——番茄和鸡蛋就变得十分重要了，所以在上面的两个问题中，低频和高频信息都有可能十分重要。因此，通过卷积层分离低频和高频使它们能够被分别处理也就变得十分重要了。

傅立叶变换及频域信息分析显然没有上面说的这么简单，更多的细节还需要更多知识和更深入地分析，这里只是“抛砖”作个引子，让读者多一种理解卷积的角度。

## 4.2 卷积层的反向传播

前面介绍了卷积层的基本算法和运算语意，下面就介绍卷积层的反向计算优化方法。一般来说，卷积层的反向传播有两种算法。

1. 实力派解法。
2. 软件库中常用的套路——“整容”后的偶像派解法。

和第 3 章介绍全连接层一样，这里也将给出一个小例子。假定输入是一个  $1 \times 5 \times 5$  的图像，卷积层的维度是  $3 \times 3$ ，同时  $\text{stride}=1$ ， $\text{padding}=0$ 。最终的输出是  $1 \times 3 \times 3$ ，那么卷积操作运算图如图 4-12 所示。

图 4-12 还包含了每一个矩阵的位置标记，其中  $\mathbf{X}$  表示输入的矩阵，位置下标为 0-24； $\mathbf{K}$  表示卷积核的矩阵，位置下标为 0-8； $\mathbf{Z}$  表示卷积的结果，位置下标为 0-8。

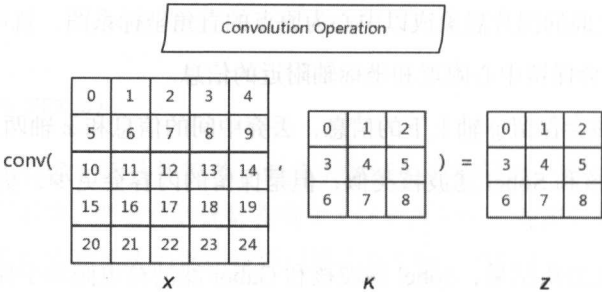


图 4-12 卷积操作示意图

4.2.1 实力派解法

所谓的实力派解法就是用卷积定义（这里就用相关操作）做前向计算，然后利用前向的算法推导反向计算。因为这个过程有点复杂，需要对卷积操作有比较深刻的理解，所以被称为实力派解法。

卷积运算的参与者共有三位，所以观察卷积操作的视角也有三种，首先是卷积操作的第一视角，它的运算形式和卷积操作的基本算法一致。运算的基本流程如图 4-13 所示。

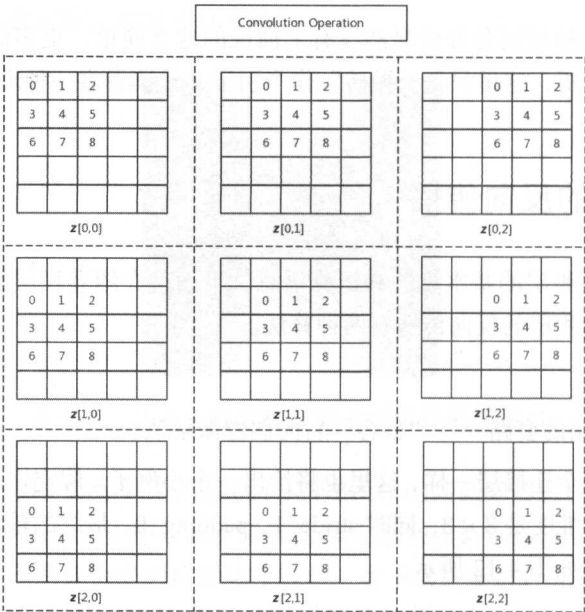


图 4-13 卷积操作视角 1：“卷积结果—运算图像—卷积核”

图 4-13 详细地讲述了前向计算的全过程，这个图的观察层次为“卷积结果—运算图像—卷积核”。其中最外面维度为  $3 \times 3$  的虚线框表示结果  $Z$ ，每一个虚线格子表示自身被卷积计算的过程，虚线格子内包含一个  $5 \times 5$  的矩阵，这个矩阵代表了运算图像  $X$ ，图像中的数字表示卷积核  $K$  的位置信息，也就是记录了卷积核和图像运算时的对应关系。比方说，卷积结果  $Z$  第 2 行（以下的下标一律从 0 开始）第 1 列的位置  $z[2,1]$  处，这里面的卷积核是和图像中的第 2 行第 1 列开始的  $3 \times 3$  的子矩阵做元素级相乘并加和，才得到了  $z[2,1]$  的结果。

了解了第一种视角，下面先来看看反向传播的计算。反向传播中偏置项 bias 的偏导数计算相对简单，这里就不说了，它和全连接层反向传播中的偏置项一样。除此之外，每一层还需要计算两类数值。

1. 卷积层输入图像  $X$  对目标函数的偏导数。
2. 卷积层线性部分参数  $W$  对目标函数的偏导数。

由于卷积操作的特殊性，输入图像和参数的运算耦合得比较严重，想要清晰地分离它们之间的运算还需要从其他视角观察卷积操作。首先来计算输入图像的偏导数。利用第 3 章中全连接层反向计算的思路，想求出每个输入的偏导数，就要列出每个输入元素参与的计算，然后用后一层计算得到的梯度与每一个输入元素运算的参数和卷积层输出的偏导相乘，最后再求和，就可以得到想要的结果。这个运算公式和全连接层的公式是一致的：

$$\frac{\partial \text{Loss}}{\partial X_i} = \left[ \frac{\partial \text{Loss}}{\partial Z_j}, \dots \right]^T \left[ \frac{\partial Z_j}{\partial K_l}, \dots \right]$$

为了完成这个运算，下面就要展示卷积操作的第二视角，这个视角的主角就是输入数据，如图 4-14 所示。

观察视角是“运算图像—卷积结果—卷积核”。最外层是  $5 \times 5$  的框，表示每一个图像数据  $X$  在卷积运算中参与的计算。每一个框里面的实线  $3 \times 3$  矩阵表示了卷积的结果  $Z$ ，其中有些位置标记了数字，有些则没有。那些标记了数字的位置代表虚线框所在位置的输入数据参与了这个结果位置的计算，而数字的内容表示了当时运算时与它配对的卷积核位置。

比方说  $x[0,0]$  的位置，由于它的位置靠边，它只和卷积核的 0 号位置进行运算，得到的结果保存在卷积结果的 0 号位置；而  $x[0,1]$  则完成了两次运算：和卷积核的 1 号位置计算，结果保存在 0 号位置；和卷积核的 0 号位置计算，结果保存在 1 号位置。对于正中间的元素，它对每一个卷积结果都做了“贡献”，因此它参与了 9 次计算，对应的卷积核编号在图中都可以看到。

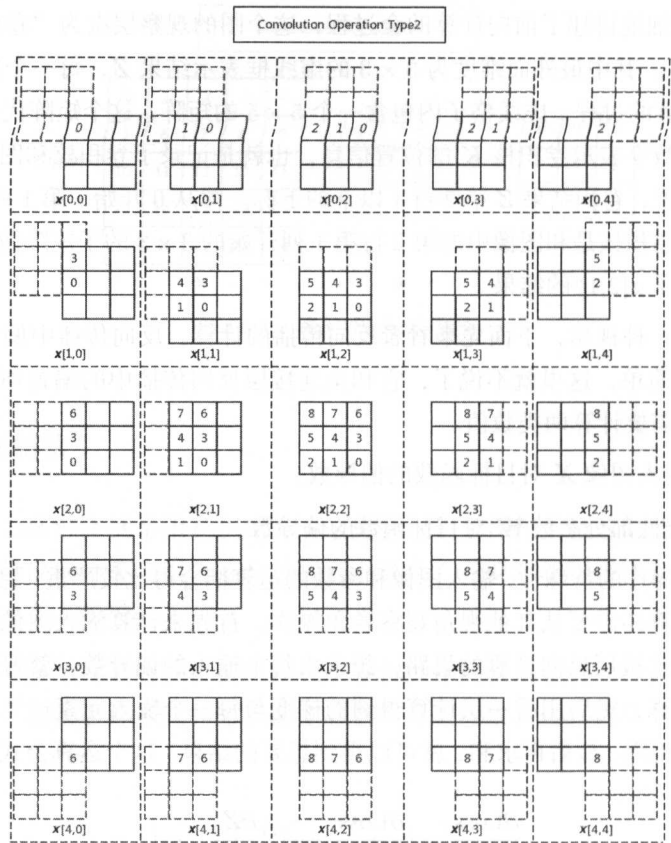


图 4-14 卷积操作视角 2：“运算图像—卷积结果—卷积核”

图 4-14 也展示出了卷积在运算时的特点：边角位置完成的运算会比中间少。如果把每一个输入位置参与的运算次数都展示出来，那么这些信息就可以组成一个矩阵。

1	2	3	2	1
2	4	6	4	2
3	6	9	6	3
2	4	6	4	2
1	2	3	2	1

这个矩阵中所有元素的和为 81，而从第一个视角出发计算的运算次数也是 81。这说明虽然观察的视角不同，但是实际上运算过程是一致的。

虽然完成了输入数据的运算分离，但是每一个位置的计算量并不相同，那么计算梯度时该如何进行呢？图 4-14 实际上也给出了一种巧妙的计算方法。由于有些元素计算时卷积核并没有完全参与，于是图中就用虚线表示了未参与运算的卷积核位置。完

成了补充之后，读者就会惊奇地发现，每一个卷积核实际上被旋转了 180 度，而且将这些小图拼接起来，实际上整张大图的运算可以用一个卷积操作表示。这个卷积操作要做的准备工作如下。

- 1. 卷积核需要旋转 180 度。
- 2. 输出数据的梯度所在的矩阵需要在周围做 padding，padding 的数量等于卷积核的维度减 1，在上面的例子中，padding=2。

所以这一步看似复杂的求解最终可以转化成一行伪代码，这样就完成了第二视角的观察和对卷积层输入偏导数的求解：

```
residual_x=conv2(padding(residual_z,kernel.shape() - 1), rot180(kernel))
```

下面要做的是第三视角和计算卷积层参数的梯度。和前面的方法类似，首先要做的是重新整理运算关系，将参数参与的运算表示出来。图 4-15 所示为第三视角的观察角度。

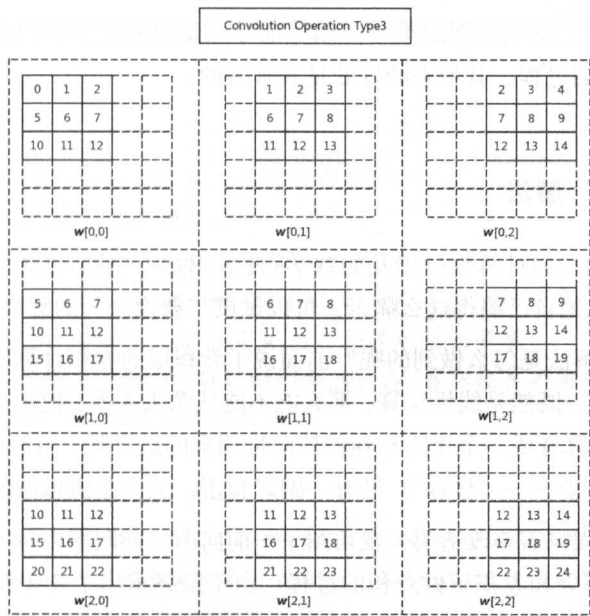


图 4-15 卷积操作视角 3：“卷积核—卷积结果—运算图像”

第三视角的观察顺序是“卷积核—运算图像—卷积结果”。其中最外层框的维度是  $3 \times 3$ ，里面的每一格都表示当前卷积核参数和图像数据参与的运算。格内黑色矩阵表示计算结果，上面的数字代表输入图像的位置信息，从图 4-15 中可以看出，每个参数都参与了所有输出位置的运算。例如  $k[0,0]$ ，由于它是卷积核的左上角，所以它都和对应的输入图像区域的左上角做运算，实际上参与运算的是那 9 个位置。



从图 4-15 可以很清楚地看出这个关系。这一次直接把输入数据和输出数据的梯度做卷积，就可以得到参数  $w$  的导数。比上一步的计算还要简单点。因此这里的求解也只需要一步就可以完成：

```
residual_w=conv2(x,residual_z)
```

到此为止，卷积层反向计算的暴力解法就结束了。从上面的分析中可以看出，想要采用暴力派的解法需要能清晰地画出下面三张图。

- 前向计算图：标有卷积核  $id$  的输入小图组成的输出大图。
- 下层 Loss：标有卷积核  $id$  的输出小图组成的输入大图。
- 本层  $w$  导数：标有输入  $id$  的输出小图组成的卷积核大图。

刚才看到的只是  $stride=1$ ,  $padding=0$  的解法，相对来说省略了一些细节的考虑。如果把这些参数加上，这种方法也是可以解的，只不过比刚才复杂一些，这里就不做更进一步的推导了，有志成为“实力派”的读者可以尝试进一步推导。但不管问题做怎么样的变换，都离不开上面三张图的推导。熟练推导三张图能够让我们更深刻地理解卷积（相关）操作内部的过程，其带来的好处是非常多的。

#### 4.2.2 “偶像派”解法

卷积层里前向后向计算推导涉及的数学问题并不多，但是想熟练掌握还需要多练习。读者如果不想把自己搞得这么痛苦，可以试试“偶像派”的解法。

“偶像派”的解法是怎么做到的呢？前面说了卷积层的卷积计算属于线性部分，当然也就是线性的了。既然是线性运算，那么能不能从源头入手，把输入图像做一个彻底的变换，使它的运算变成一个矩阵和卷积核向量相乘的运算呢？当然可以了，这就涉及偶像派解法的第一步——“整容”。没有一副好脸蛋，怎么走偶像派的路子？

“整容”的过程只需要改造第一视角图——前向图。前向图中每一个小图都可以表示输入数据的一部分和卷积核做点积的过程。由于最终求出了 9 个结果值，因此也就有 9 个输入的部分数据和卷积做了点积。基于这个思想，输入矩阵就可以变成如图 4-16 所示的样子。

完成了变换后，前向后向计算就都可以用全连接层的算法计算了，全连接层的思路比卷积层更清晰，在完成计算的同时也可以放松大脑。同样地，如果考虑了  $padding$  和  $stride$ ，写这个转换的算法还是需要多一些考虑。这里给出开源框架 Caffe（第 5 章中将介绍）中有关于这部分功能的实现算法，为了阅读方便在此删减了部分代码：

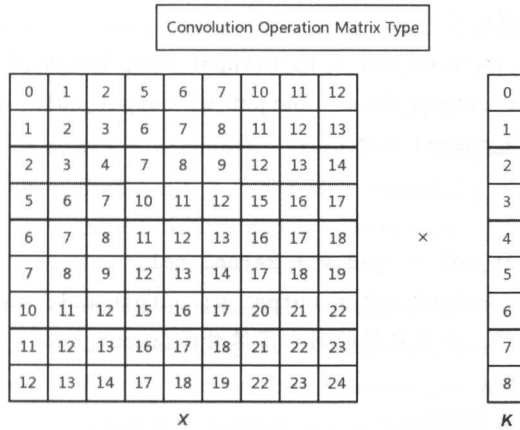


图 4-16 矩阵形式的卷积操作

```

inline bool is_a_ge_zero_and_a_lt_b(int a, int b) {
    return static_cast<unsigned>(a) < static_cast<unsigned>(b);
}

void im2col_cpu(
    const Dtype* data_im, // 输入的图像数据
    const int channels, // 通道数
    const int height, const int width, // 图像的长宽
    const int kernel_h, const int kernel_w, // 卷积核的长宽
    const int pad_h, const int pad_w, // 图像的填充长宽
    const int stride_h, const int stride_w, // 卷积计算的间隔长宽
    Dtype* data_col // 最终输出的数据
) {
    // 首先计算输出的维度：长、宽
    const int output_h = (height + 2 * pad_h - kernel_h) / stride_h + 1;
    const int output_w = (width + 2 * pad_w - kernel_w) / stride_w + 1;
    const int channel_size = height * width;
    // 对于每一个通道
    for (int channel = channels; channel--; data_im += channel_size) {
        for (int kernel_row = 0; kernel_row < kernel_h; kernel_row++) {
            int input_row = -pad_h + kernel_row;
            for (int kernel_col = 0; kernel_col < kernel_w; kernel_col++) {
                for (int output_rows = output_h; output_rows; output_rows--) {
                    // 如果转换的行在图像外面(input_row < 0 || input_row > height),

```

```

// 就直接填0
if (!is_a_ge_zero_and_a_lt_b(input_row, height)) {
    for (int output_cols = output_w; output_cols; output_cols--) {
        *(data_col++) = 0;
    }
} else {
    int input_col = -pad_w + kernel_col;
    for (int output_col = output_w; output_col; output_col--) {
        // 同理，如果转换的列在图像外面(input_col < 0 || input_col >
        width),
        // 就直接填0
        if (is_a_ge_zero_and_a_lt_b(input_col, width)) {
            *(data_col++) = data_im[input_row * width + input_col];
        } else {
            *(data_col++) = 0;
        }
        input_col += stride_w;
    }
}
input_row += stride_h;
}
}
}
}
}
}
}

```

前面介绍的方法是将图像矩阵放在乘法计算的左边，而 Caffe 中的代码如果不进行转置，是将图像矩阵放在右边。它的维度是  $(kernel_h \cdot kernel_w)(output_h \cdot output_w)$ ，矩阵的每一行存储的是和卷积核某一个参数相乘的所有图像数据，每一列存储的是一个卷积子操作需要的数据。了解了这些再看这段代码就会简单许多。

看完了从原始图像到矩阵的代码，再看看从运算矩阵到图像的过程：

```

void col2im_cpu(
    const Dtype* data_col, // 运算矩阵
    const int channels, // 通道数
    const int height, const int width, // 图像长宽
    const int kernel_h, const int kernel_w, // 卷积核长宽
    const int pad_h, const int pad_w, // 填充长宽

```

```

    const int stride_h, const int stride_w, // 卷积计算的长宽
    Dtype* data_im // 图像输出
) {
    caffe_set(height * width * channels, Dtype(0), data_im);
    const int output_h = (height + 2 * pad_h - kernel_h) / stride_h + 1;
    const int output_w = (width + 2 * pad_w - kernel_w) / stride_w + 1;
    const int channel_size = height * width;
    for (int channel = channels; channel--; data_im += channel_size) {
        for (int kernel_row = 0; kernel_row < kernel_h; kernel_row++) {
            for (int kernel_col = 0; kernel_col < kernel_w; kernel_col++) {
                int input_row = -pad_h + kernel_row * dilation_h;
                for (int output_rows = output_h; output_rows; output_rows--) {
                    if (!is_a_ge_zero_and_a_lt_b(input_row, height)) {
                        data_col += output_w;
                    } else {
                        int input_col = -pad_w + kernel_col * dilation_w;
                        for (int output_col = output_w; output_col; output_col--) {
                            if (is_a_ge_zero_and_a_lt_b(input_col, width)) {
                                data_im[input_row * width + input_col] += *data_col;
                            }
                            data_col++;
                            input_col += stride_w;
                        }
                    }
                    input_row += stride_h;
                }
            }
        }
    }
}

```

看完了上面从图像到计算矩阵的代码，这个反向的代码也就比较简单了。

这样，“偶像派”的解法也就完成了。和实力派比起来是不是挺简单的？借助了矩阵和向量的乘法，读者的大脑得到了极大的解放。了解了两种算法，读者自然会想到一个问题——那些开源的运算框架是如何实现卷积层的运算的呢？

基本上开源框架都选择了“偶像”的道路……倒不是因为偶像派的思路清晰简单，而是因为矩阵乘法这样的运算经过多年的研究，运算效率非常有保障。卷积运算在实

现过程中会因为 cache 友好性等问题导致性能较差，所以最终被淘汰。从上面 Caffe 的代码中可以感受到，从图像到运算矩阵转换的代码效率不会太高。虽然解放了大脑，但是这样写代码不见得能提高多少速度。实际上，Caffe 内部最快速的方法调用了现在被广大群众所爱戴的——cuDNN 库的实现，这个方法比上面提到的“偶像派”解法在细节上更精细，可以算是偶像与实力兼备。感兴趣的读者可以深入了解一下。

### 4.3 ReLU

前面介绍了卷积层的线性部分，下面开始介绍非线性部分。在第 3 章读者就已经了解过两个非线性部分的函数：

- Sigmoid
- Tanh

随着深度网络的发展，科研人员又发现了另一个非常好用的非线性函数，那就是 ReLU，全称 Rectify Linear Unit。它的函数形式是这样的：

```
def relu(x):  
    return x if x > 0 else 0
```

函数的形状如图 4-17 所示。

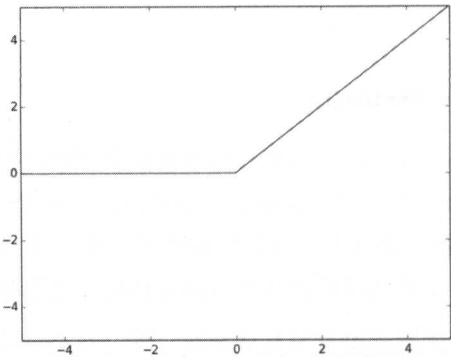


图 4-17 ReLU 函数的图

和前面的两个函数比起来，这个函数看上去十分简单，似乎只需要小学数学就可以明白，但它却是现在使用量最高的非线性函数。以它为基础，科研人员还研究出了一系列变形函数，大有统治非线性函数界的势头。前面两个函数已经称霸江湖许多年了，为什么一个新来的简单到不行的函数会抢尽风头呢？

### 4.3.1 梯度消失问题

梯度消失是深层模型中特有的问题，这其中要数 Sigmoid 函数做得差了。早期的神经网络主要是浅层神经网络，在反向传导的过程中，因为中间经过的网络层不多，所以从前面的网络回传的残差基本还算是“新鲜”的。随着深度学习的发展，网络层数的不断加深，反向传导逐渐变成了一个“漫长”的事情。从最前端的目标函数损失开始向后传导，一路上会有各种各样的数据改变梯度的数量，等到了后面的网络“手”上，这些被加工后的残差有时会变得面目全非。

那么是怎样的面目全非法呢？这主要体现在数值的范围上。两种经典的非线性函数的求导公式如下所示：

Sigmoid:  $\frac{\partial f(x)}{\partial x} = f(x)(1 - f(x))$ ，其中函数值的范围是 (0, 1)

Tanh:  $\frac{\partial f(x)}{\partial x} = 1 - f(x)^2$ ，其中函数值的范围是 (-1, 1)

把这两个导数函数画出来，如图 4-18 所示。

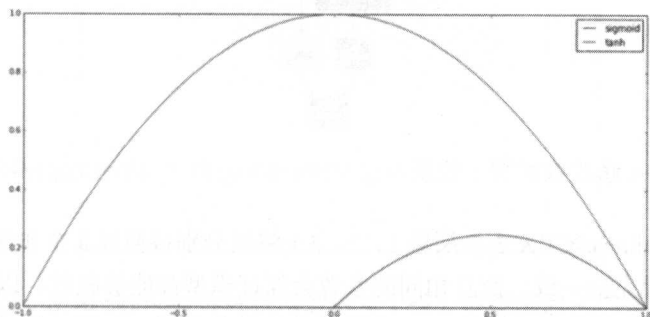


图 4-18 Sigmoid 与 Tanh 的导数图

可以看出，Tanh 的导数数值虽然没有大于 1，但是与 1 的距离偏差得不算明显；而 Sigmoid 就比较可怜了，最大也就只有 0.25。从前面介绍反向计算的章节中已经知道，当导数通过非线性部分到达后面的线性部分时，非线性的函数同样会改变导数的数值，图 4-18 中显示的数量就是它们改变的数量。

这时 Sigmoid 的弱点就暴露出来了，因为它在最好情况下也会把传递的导数数值除以 4。对于不深的网络，这点儿损失算不上什么，但是对于深层的神经网络，这样的打折将会产生巨大的灾难：上层的网络得到的梯度是比较大的，下层网络得到的梯度明显小很多。这种别人吃肉我喝汤的事情自然会让底层网络发育不良，很容易让深层网络无法发挥出应有的效果。

为了验证这个事情可能带来的后果，这里使用 Caffe 中的案例模型——MNIST 的求解模型 LeNet 作为实验的对象。MNIST 数据集相信读者已经了解，模型的结构如图 4-19 所示。

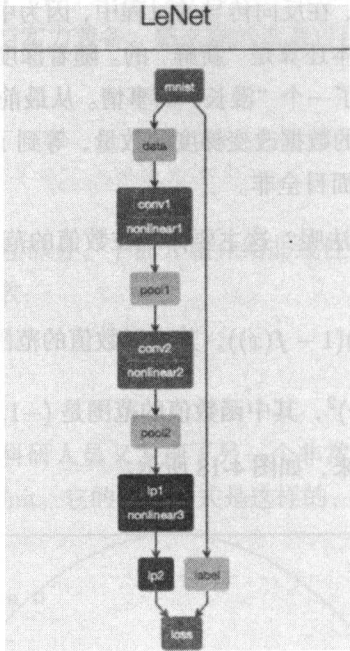


图 4-19 LeNet 修改版模型（使用 <http://ethereon.github.io/netscope/#/editor> 制作）

其中的 NonlinearX（X 表示层数 1，2，3）将被分别替换成 3 个非线性函数。模型优化的参数配置完全一致，而且相同的参数会保证模型都能够收敛。以下是 3 个模型的训练结果，首先是训练数据的 Loss 和迭代轮数的关系图，如图 4-20 所示。

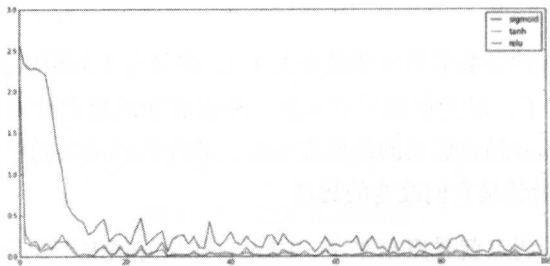


图 4-20 使用不同非线性函数时训练迭代数与 Loss 的关系

可以看出，在这个问题上，Sigmoid 明显弱一些，Tanh 和 ReLU 相近。当然，到最后 Sigmoid 与另外两者的差距也不算大。

接下来是测试集的 Loss，如图 4-21 所示。

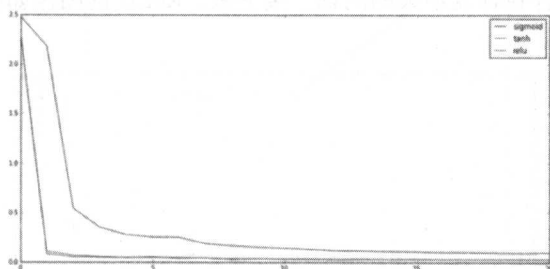


图 4-21 使用不同非线性函数时测试集的 Loss 情况

这个图的效果和前面的训练图类似。到这里基本可以断定：对于这个问题，在其他变量保持一致的情况下，使用 Sigmoid 的非线性函数训练效果会弱些。那么，这个弱一些的现象和上面提到的梯度消失问题会共存吗？

接下来将展示模型中 4 个主要层次的参数梯度的大小。首先是 Sigmoid，如图 4-22 所示。

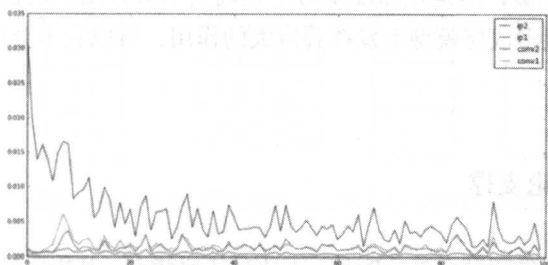


图 4-22 使用 Sigmoid 非线性函数时 4 个核心模型层的平均参数梯度

从图 4-23 中可以明显看出一个规律，那就是越靠近后面的网络层参数更新的量越大，越靠近前面的网络层参数更新的量越小。这和刚才的猜想是一致的。那么，其他的函数表现得如何呢？接下来是 Tanh 的图像，如图 4-23 所示。

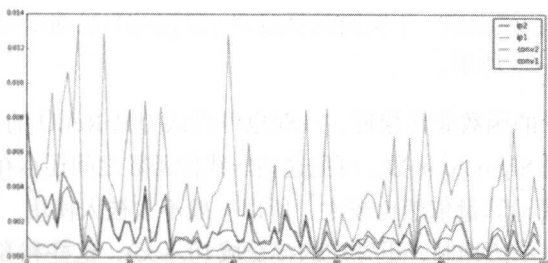


图 4-23 使用 Tanh 非线性函数时 4 个核心模型层的平均参数梯度



可以看出 Tanh 的参数梯度不再像 Sigmoid 那样满足梯度消失的规律，这也说明在这个问题中 Tanh 不存在梯度消失的问题。接下来，ReLU 的图像如图 4-24 所示。

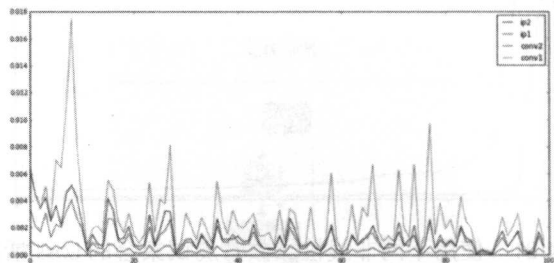


图 4-24 使用 ReLU 非线性函数时 4 个核心模型层的平均参数更新量

ReLU 的结果同样没有出现梯度消失的问题，而且它的 4 条线距离和 Tanh 相比更相近一些。

上面的实验结果确实反映出 Sigmoid 在梯度消失方面的弱点，虽然它的函数曲线和人的神经反应很相似，但是在深度学习上它确实不太好用。这也是件无可奈何的事情，不过它还是在很多浅层模型上发挥着巨大的作用，所以它不会离开大家的视线。

### 4.3.2 ReLU 的理论支撑

除了解决梯度消失问题之外，ReLU 还具有其他特点，其中之一就是它的计算十分简单。与 Sigmoid、Tanh 相比，ReLU 的计算确实比较简单。虽然它的计算比较简单，但实际上 ReLU 也有很强的理论支撑。曾经有科学家研究脑神经元接受信号的激活模型，并从中总结了一系列的特点，经过研究发现一个称为 Softplus 的函数恰好拥有这些特点：

$$\text{Softplus}(x) = \log(1 + e^x)$$

它的图像如图 4-25 所示。

由于它和 ReLU 的函数非常相近，一般也可以认为是 ReLU 的平滑版。更巧的是，它的导数函数刚好是 Sigmoid 函数，可见这些非线性函数之间还存在着一定的关系。然而这个函数相对来说计算量依然比较大，所以一般不太被人使用。不过它代表了激活函数的一个新方向：单边抑制，更宽广的接受域  $(0, +\infty)$ ，这些和 Sigmoid 函数相比有很大的不同。

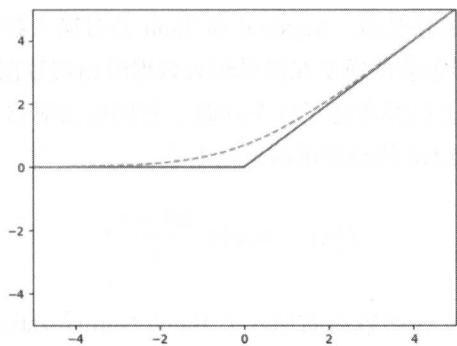


图 4-25 Softplus 函数和 ReLU 函数的对比图，虚线为 Softplus，实线为 ReLU

### 4.3.3 ReLU 的线性性质

作为一个非线性函数，ReLU 实际上还具有其他非线性函数所不具备的线性性质，很多科研人员也都提到了这一点。因为它强行把小于 0 的部分截断，那么对于线性部分的输出，最终的结果就好像左乘一个非 0 即 1 的对角阵，如图 4-26 所示。

$$\text{ReLU}\left(\begin{bmatrix} 0.4 \\ 0.2 \\ -0.4 \end{bmatrix}\right) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \times \begin{bmatrix} 0.4 \\ 0.2 \\ -0.4 \end{bmatrix} = \begin{bmatrix} 0.4 \\ 0.2 \\ 0 \end{bmatrix}$$

图 4-26 ReLU 可以被想象成线性变换的形式

这样看来它也十分像一个线性层。在第 3 章中介绍全连接层的非线性部分时，曾经介绍过非线性部分的一大作用——打破线性关系，这一点 ReLU 是可以做到的，但是与此同时，它仍可以被看作是一个线性操作，这对很多分析深层模型理论的科研人员来说实在是一个福音，因为线性操作的分析难度稍微简单一些。如果把非线性层的输入想象成高维空间的一个向量，那么 ReLU 就是完成一次向量的投影，舍弃了其中的负数部分。从这个角度来思考，深层神经网络的计算过程就变成了“线性变换—投影—线性变换—投影”这样不断循环的过程，在这样的框架下，相信深层模型的理论会更容易被研究清楚。

### 4.3.4 ReLU 的不足

虽然 ReLU 有很多优点，但是不可否认它依然不是一个完美的非线性函数。可以简单介绍它在两个方面的问题。

首先是它过于宽广的接受域。Sigmoid 和 Tanh 会对输入数据的上界进行限制，而 *ReLU 则完全不会，这样也会使模型在接受较大数据时出现数据不稳定。实际上，有一些第三方函数库在实现上已经考虑了这个问题，它们也会对这个函数的上界做一个限定——于是一个叫做 ReLU6 的经验函数就诞生了：*

$$f(x) = \min(6, \frac{|x| + x}{2})$$

更有一些网络在 ReLU 函数前面加入了 Batch Normalization 这样的归一化层来弥补它的不足。总而言之，限定函数的范围对于我们的计算是一件好事儿，它帮助我们解决了很多数值上的问题。关于这部分的问题，将在第 6 章详细介绍。

ReLU 的另一个问题来自它的负数方向。对负数的完全抑制使得它像 Sigmoid 一样促使很多神经元参数无法得到有效的更新，因为一旦 ReLU 的输入是负数，那么 ReLU 会将它置为 0，同时这个输入数据的梯度也将为 0，这样从这个数据出发的反向计算将全部为 0。如果随着训练过程不断进行，这个现象一直得不到改变，那么这个数据就好像死掉了一样，这就是“Dying ReLU problem”的由来。为了解决这个问题，科研人员发明了一系列的改进函数，像 Leaky ReLU、pReLU 等，这些改进这里就不赘述了。

## 4.4 总结

本章主要介绍了卷积层相关的内容，让我们回顾一下。

- 卷积操作：局部区域相乘，stride、padding 参数对卷积的影响。
- 卷积的语意：局部区域数据相关性计算，高频低频等频率信息解耦。
- 反向传播：实力派解法——观察卷积操作的三个角度；偶像派解法——“整容”后使用和全连接层相同的方法计算。
- ReLU 非线性函数：梯度不易消失，算法简单，具有线性性质。

## 4.5 参考文献

[1] Gabor Filter [http://en.wikipedia.org/wiki/Gabor\\_filter](http://en.wikipedia.org/wiki/Gabor_filter)

# 5

## Caffe 入门

经过 3、4 两章对全连接层和卷积层的介绍，相信读者对神经网络已经有了一定的概念。很多人说深度学习是一门玄学，好比修仙炼丹，把材料扔进丹炉，不管里面发生了什么，做出想要的东西就好。这个比喻也有几分道理。对我们来说，读书领悟固然很重要，但亲身实战去炼炼丹更是必不可少。于是接下来的首要目标是找一个炼丹炉，继续深度学习的探索之旅。

当然，这里所说的炼丹炉就是一个可以运行深度学习应用的软件框架。现在学术科研界已经涌现出许许多多的优秀开源软件，本章的主要介绍对象是深度学习中鼎鼎大名的开源框架 Caffe，目标是带领读者一起熟悉它的使用方法，了解它的内容原理，掌握修改框架和增加功能的方法流程。

目前市面上有很多优秀的开源代码，为什么要选择并介绍 Caffe？Caffe 有什么优点？Caffe 具有很多优秀的特性，最大的特点就是它的代码整体上相对简单，可读性很好，架构比较清晰，不像其他开源框架拥有复杂的机制；同时功能又足够全面，满足工业级应用的需求。读者会在本章体会到这些特点。了解这些基础内容对后面的工作很有帮助，因为后面的章节还会利用 Caffe 做很多实验，了解更多深度模型内部的机理，花一些时间打好基础十分重要。

本章先回顾一个使用 Caffe 进行深层模型训练的全过程。整个过程从创建训练数据的数据库开始，经历了训练配置的编写、训练与再训练、训练过程分析、验证等过程。从这个过程中也可以看出，Caffe 基本上可以很好地涵盖模型训练的所有功能。

接下来将详细地回顾 Caffe 中较为经典的模型层的配置参数，并简单地介绍之前没提到的几个模型层。

完成了对框架基本使用的了解，下面的步骤就是深入框架内部了解 Caffe 源码中的整体架构使读者更好地理解其中的设计原理，也可以更好地完成代码修改的工作。本章将按顺序介绍每一个具体模块的内容——模型层（Layer）、网络结构（Net）、求解器（Solver）、数据层（Data Layer）和数据处理器（Data Transformer）。掌握这些模块的细节并不是必需的，但在平时的使用中，一旦遇到异常情况，这些细节能够帮助读者更快速地定位问题、解决问题。

以上对源码的分析是为最后的代码扩展服务的。本章将用一个简单的例子展示扩展 Caffe 代码的基本套路。了解了这个套路，读者就可以自行开发，让 Caffe 变得强大无比。

## 5.1 使用 Caffe 进行深度学习训练

本节的目标是使用 Caffe 完成深度学习训练的全过程。由于本章介绍的内容都是 Caffe 中成型很久的功能，绝大多数版本的 Caffe 都包含这些功能，因此读者选择一个近期的 Caffe 版本即可。关于 Caffe 下载和安装的内容请读者参考 Caffe 官方网站的指导，这里就不再赘述了。

一个常规的监督学习任务主要包含训练与预测两个大的步骤，这里还是以 Caffe 中自带的例子——MNIST 数据集手写数字识别为例（这个例子还会在后面反复见到），介绍它具体的使用方法。

如果把上面提到的深度学习训练任务分解得更细致一些，那么这个常规流程可以分成如下几个步骤。

1. 数据预处理（建立数据库）。
2. 网络结构与模型训练的配置。
3. 训练与再训练。
4. 训练日志分析。
5. 预测检验与分析。
6. 性能测试。

下面就来一一介绍。

### 5.1.1 数据预处理

首先是训练数据和预测数据的预处理。这里的工作一般是把待分析识别的图像进行简单的预处理，然后保存到数据库中。为什么要完成这一步而不是直接从图像文件

中读取数据呢？因为实际任务中，训练数据的数量可能非常大，从图像文件中读取数据并进行初始化的效率是非常低的，所以很有必要把数据预先保存在数据库中來加快训练的節奏。

以下操作將全部在终端完成。第一步是將数据下载到本地，好在 MNIST 的数据量不算大，如果读者的网络环境好，这一步的速度会非常快。首先來到 Caffe 的安装根目录——CAFFE\_HOME，然后执行下面的命令：

```
cd data/mnist
./get_mnist.sh
```

程序执行完成后，文件夹下应该会多出來 4 个文件，这 4 个文件就是我们下载的数据文件。第二步，我们需要调用 examples 中的数据库创建程序：

```
cd $CAFFE_HOME
./examples/mnist/create_mnist.sh
```

程序执行完成后，examples/mnist 文件夹下面就会多出两个文件夹，分别保存了 MNIST 的训练和测试数据。值得一提的是，数据库的格式可以通过修改脚本的 BACK-END 变量更换。目前，数据库有以下两种主流选择。

- LevelDB
- Lmdb

这两种数据库在存储数据和操纵上有些不同，首先是它们的数据组织方式不同，LevelDB 的内容如下：

```
ls examples/mnist/mnist_train_leveldb/
000014.sst  000017.sst  000021.log  000024.sst  LOG
000015.sst  000019.sst  000022.sst  CURRENT     MANIFEST-000002
000016.sst  000020.sst  000023.log  LOCK
```

LMDB 的内容如下：

```
ls examples/mnist/mnist_train_lmdb/
data.mdb  lock.mdb
```

从结构可以看出，LevelDB 的文件比较多，LMDB 的文件更紧凑。

其次是它们读取数据的接口，某些场景需要遍历数据库完成一些原始图像的分析处理，因此了解它们的数据读取方法十分必要。首先是 LMDB 读取数据的代码：

```
import numpy as np
import lmdb
```

```
import sys
import caffe
from caffe.proto import caffe_pb2

def lmdb_process(db_path):
    env = lmdb.open(db_path)
    datum = caffe_pb2.Datum()
    item_id = 0
    with env.begin() as txn:
        cursor = txn.cursor()
        for key, value in cursor:
            datum.ParseFromString(value)
            label = datum.label
            img = caffe.io.datum_to_array(datum)
            # do something here
            item_id += 1
    print item_id

if __name__ == '__main__':
    db_path = sys.argv[1]
    lmdb_process(db_path)
```

其次是 LevelDB 读取数据的代码：

```
import sys
import caffe
from caffe.proto import caffe_pb2
import leveldb
import numpy as np
from skimage import io

def leveldb_process(path):
    db = leveldb.LevelDB(path)
    datum = caffe_pb2.Datum()

    item_id = 0
    for key,value in db.RangeIter():
        datum.ParseFromString(value)
```

```

        label = datum.label
        data = caffe.io.datum_to_array(datum)
        # do something here
        item_id += 1
    print item_id

if __name__ == '__main__':
    path = sys.argv[1]
    leveldb_process(path)

```

最后回到本节的问题：为什么要采用数据库的方式存储数据而不是直接读取图像？这里可以简单测试一下用 MNIST 数据构建的这两个数据库按序读取的速度，用系统函数 time 计时，结果如下：

```
$ time python read_lmdb.py mnist_train_lmdb/
```

#以下为运行结果显示

```
60000
```

```
real    0m1.523s
```

```
user    0m1.439s
```

```
sys 0m0.085s
```

```
$ time python read_leveldb.py mnist_train_leveldb/
```

#以下为运行结果显示

```
60000
```

```
real    0m1.706s
```

```
user    0m1.609s
```

```
sys 0m0.096s
```

为了比较原始图像读入的速度，这里将 MNIST 的数据以 jpeg 的格式保存成图像，并测试它的读取效率（以 Caffe python 使用的 scikit image 为例），代码如下所示：

```
from skimage import io
```

```
import sys
```

```
import os
```

```
folder = sys.argv[1]
```



```
l = os.listdir(folder)

item_id = 0
for item in l:
    img = io.imread(folder + item, as_grey=True)
    item_id += 1
print item_id
```

最终的时间如下所示：

```
$ time python read_ori.py mnist_train_ori/
#以下为运行结果显示
60000

real    0m9.411s
user    0m8.345s
sys 0m1.069s
```

由此可以看出，原始图像和数据库相比，读取数据的效率差距很大。虽然在 Caffe 训练中数据读入是异步完成的（5.7 节会介绍），但是它还是不能太慢，所以这也就是在训练时选择数据库的原因。

至于这两个数据库之间的比较，本节就不分析了。感兴趣的读者可以在一些大型数据集上做一些实验，那样更容易看出两个数据集之间的区别。

## 5.1.2 网络结构与模型训练的配置

5.1.1 节完成了数据库的创建，下面就要为训练模型做准备了。一般来说，Caffe 采用读入配置文件的方式进行训练。Caffe 的配置文件一般由两部分组成：*solver.prototxt* 和 *net.prototxt*（有时会有多个 *net.prototxt*）。它们实际上对应了 Caffe 系统架构中两个十分关键的实体——网络结构（Net）和求解器（Solver）。先来看看一般来说相对简短的 *solver.prototxt* 的内容，为了方便读者理解，所有配置信息都已经加入了注释。

```
# 告诉我们网络结构配置在哪儿
net: "examples/mnist/lenet_train_test.prototxt"
# 我们要用GPU训练
solver_mode: GPU
# 这次的训练一共有10000次迭代，每次迭代跑多大数据？net里面会给我们答案
max_iter: 10000
```

```
# 这次的测试每500轮跑一遍，一遍跑100个迭代
test_iter: 100
test_interval: 500
# 每100轮为我们输出一些信息
display: 100
# 基础的学习率是0.01，学习率的衰减形式后面再说
base_lr: 0.01
lr_policy: "inv"
gamma: 0.0001
power: 0.75
# 动量衰减率是0.9，正则项的权重是0.0005
momentum: 0.9
weight_decay: 0.0005
# 每5000轮保存一下进度
snapshot: 5000
snapshot_prefix: "examples/mnist/lenet"
```

为了方便读者理解，本书将 *examples/mnist/lenet\_solver.prototxt* 中的内容进行重新排序，整个配置文件相当于回答了下面几个问题。

- 网络结构的文件在哪儿？
- 用什么计算资源训练？CPU 还是 GPU？
- 训练多久？训练和测试的比例是如何安排的，什么时候输出些给我们瞧瞧？
- 优化的学习率怎么设定？其他优化参数——如动量和正则呢？
- 要时刻记得存档，不然得从头来过。

接下来就是 *net.prototxt* 了，这里忽略了每个网络层的参数配置，只把表示网络的基本结构和类型配置展示出来：

```
name: "LeNet"
layer {
  name: "mnist"
  type: "Data"
  top: "data"
  top: "label"
}
layer {
  name: "conv1"
```

```
    type: "Convolution"
    bottom: "data"
    top: "conv1"
}
layer {
    name: "pool1"
    type: "Pooling"
    bottom: "conv1"
    top: "pool1"
}
layer {
    name: "conv2"
    type: "Convolution"
    bottom: "pool1"
    top: "conv2"
}
layer {
    name: "pool2"
    type: "Pooling"
    bottom: "conv2"
    top: "pool2"
}
layer {
    name: "ip1"
    type: "InnerProduct"
    bottom: "pool2"
    top: "ip1"
}
layer {
    name: "relu1"
    type: "ReLU"
    bottom: "ip1"
    top: "ip1"
}
layer {
    name: "ip2"
    type: "InnerProduct"
```

```

    bottom: "ip1"
    top: "ip2"
}

```

```

layer {
    name: "loss"
    type: "SoftmaxWithLoss"
    bottom: "ip2"
    bottom: "label"
    top: "loss"
}

```

本节只展示了网络结构的基础配置，但也占用了大量的篇幅。一般来说，这个文件中的内容超过 100 行都是常见的事。像大名鼎鼎的 ResNet 网络，它的文件长度通常在千行以上，更是让人难以阅读。那么问题来了，那么大的网络文件都是靠人直接编辑出来的么？不一定。有的人会比较有耐心地一点点写完，而有的人则不愿意做这样的苦活。实际上，Caffe 提供了一套接口，读者可以通过写代码的形式生成这个文件。这样一来，编写模型配置的工作也变得简单不少。下面展示了一段生成 LeNet 网络结构的代码：

```

import sys

import caffe

from caffe import layers as L
from caffe import params as P

class LeNet(object):
    def __init__(self, lmbd_train, lmbd_test, num_output):
        self.train_data = lmbd_train
        self.test_data = lmbd_test
        self.class_num = num_output

    def lenet_proto(self, batch_size):
        n = caffe.NetSpec()
        n.data, n.label = L.Data(
            source=self.train_data,
            backend=P.Data.LMDB,
            batch_size=batch_size,

```

```

        ntop=2,
        transform_param=dict(scale=0.00390625, mirror=False))
n.conv1 = L.Convolution(n.data,
    kernel_size=5, num_output=20, stride=1,
    weight_filler=dict(type='xavier'),
    bias_filler=dict(type='constant'))
n.pool1 = L.Pooling(n.conv1,
    pool=P.Pooling.MAX, kernel_size=2, stride=2)
n.conv2 = L.Convolution(n.pool1,
    kernel_size=5, num_output=50, stride=1,
    weight_filler=dict(type='xavier'),
    bias_filler=dict(type='constant'))
n.pool2 = L.Pooling(n.conv2,
    pool=P.Pooling.MAX, kernel_size=2, stride=2)
n.ip1 = L.InnerProduct(n.pool2, num_output=500,
    weight_filler=dict(type='xavier'),
    bias_filler=dict(type='constant'))
n.relu1 = L.ReLU(n.ip1, in_place=True)
n.ip2 = L.InnerProduct(n.relu1,
    num_output=self.class_num,
    weight_filler=dict(type='xavier'),
    bias_filler=dict(type='constant'))
n.loss = L.SoftmaxWithLoss(n.ip2, n.label)
return n.to_proto()

if __name__ == '__main__':
    l = LeNet('123', '234', 10)
    print l.lenet_proto(128)

```

最终生成的结果如下所示：

```

layer {
  name: "data"
  type: "Data"
  top: "data"
  top: "label"
  transform_param {
    scale: 0.00390625

```

```

    mirror: false
}
data_param {
    source: "123"
    batch_size: 128
    backend: LMDB
}
}
layer {
    name: "conv1"
    type: "Convolution"
    bottom: "data"
    top: "conv1"
    convolution_param {
        num_output: 20
        kernel_size: 5
        stride: 1
        weight_filler {
            type: "xavier"
        }
        bias_filler {
            type: "constant"
        }
    }
}
}
layer {
    name: "pool1"
    type: "Pooling"
    bottom: "conv1"
    top: "pool1"
    pooling_param {
        pool: MAX
        kernel_size: 2
        stride: 2
    }
}
}
layer {

```

```

name: "conv2"
type: "Convolution"
bottom: "pool1"
top: "conv2"
convolution_param {
  num_output: 50
  kernel_size: 5
  stride: 1
  weight_filler {
    type: "xavier"
  }
  bias_filler {
    type: "constant"
  }
}
}

layer {
  name: "pool2"
  type: "Pooling"
  bottom: "conv2"
  top: "pool2"
  pooling_param {
    pool: MAX
    kernel_size: 2
    stride: 2
  }
}

layer {
  name: "ip1"
  type: "InnerProduct"
  bottom: "pool2"
  top: "ip1"
  inner_product_param {
    num_output: 500
    weight_filler {
      type: "xavier"
    }
  }
}

```

```

    bias_filler {
      type: "constant"
    }
  }
}
layer {
  name: "relu1"
  type: "ReLU"
  bottom: "ip1"
  top: "ip1"
}
layer {
  name: "ip2"
  type: "InnerProduct"
  bottom: "ip1"
  top: "ip2"
  inner_product_param {
    num_output: 10
    weight_filler {
      type: "xavier"
    }
    bias_filler {
      type: "constant"
    }
  }
}
layer {
  name: "loss"
  type: "SoftmaxWithLoss"
  bottom: "ip2"
  bottom: "label"
  top: "loss"
}

```

有些读者可能觉得上面的代码并没有节省太多篇幅，实际上，如果将上面的代码模块化做得更好些，它就会变得非常简洁。这里就不做演示了，欢迎读者自行尝试。



### 5.1.3 训练与再训练

准备好了数据，也确定了训练相关的配置，下面正式开始训练。训练需要启动这个脚本：

```
./examples/mnist/train_lenet.sh
```

然后经过一段时间的训练，命令行产生了大量日志，训练过程也宣告完成。这时训练好的模型目录多出了这几个文件：

```
lenet_iter_10000.caffemodel
lenet_iter_10000.solverstate
lenet_iter_5000.caffemodel
lenet_iter_5000.solverstate
```

很显然，这几个文件保存了训练过程中的一些内容，那么它们都是做什么的呢？caffemodel 文件保存了 caffe 模型中的参数，solverstate 文件保存了训练过程中的一些中间结果。保存参数这件事很容易想象，但是保存训练中的中间结果就有些抽象了。solverstate 里面究竟保存了什么？回答这个问题就需要找到 solverstate 的内容定义，这个定义来自 `src/caffe/proto/caffe.proto` 文件（这个文件也是 Caffe 中的一大核心文件，5.2 节会详细介绍）：

```
// A message that stores the solver snapshots
message SolverState {
    optional int32 iter = 1; // The current iteration
    optional string learned_net = 2; // The file that stores the learned net
    .
    repeated BlobProto history = 3; // The history for sgd solvers
    optional int32 current_step = 4 [default = 0]; // The current step for
    learning rate
}
```

从定义中可以很清楚地看出其内容的含义。其中 history 是一个比较有意思的信息，他存储了历史的参数优化信息。这个信息有什么作用呢？由于很多算法都依赖历史更新信息（例如后面会接触到的动量算法），如果有一个模型训练了一半停止了，现在想基于之前训练的成果继续训练，那么需要历史的优化信息帮助继续训练。如果模型训练突然中断训练而历史信息又丢失了，那么模型只能从头训练。这样的深度学习框架就不具备“断点训练”的功能，只有“重头再来”的功能。现在的大型深度学习模型都需要很长的时间训练，有的需要训练好几天，如果框架不提供断点训练的功能，一旦机

器出现问题导致程序崩溃，模型就不得不重头开始训练，这会对工程师的身心造成巨大打击……所以这个存档机制极大地提高了模型训练的可靠性。

从另一方面考虑，如果模型训练彻底结束，这些历史信息就变得无用了。caffemodel 文件需要保存下来，而 solverstate 文件可以被直接丢弃。因此，这种分离存储的方式特别方便操作。

从刚才提到的“断点训练”可以看出，深度学习其实包含了“再训练”这个概念。一般来说“再训练”包含两种模式，其中一种就是上面提到的“断点训练”。从前面的配置文件中可以看出，训练的总迭代轮数是 10000 轮，每训练 5000 轮，模型就会被保存一次。如果模型在训练的过程中被一些不可抗力打断了（比方说机器断电了），那么读者可以从 5000 轮迭代时保存的模型和历史更新参数恢复出来，命令如下所示：

```
./build/tools/caffe train -solver examples/mnist/lenet_solver.prototxt -
snapshot examples/mnist/lenet_iter_5000.solverstate
```

这里不妨再进行深入分析。虽然模型的历史更新信息被保存下来了，但当时的训练场景真的被完全恢复了吗？似乎没有，还有一个影响训练的关键因素没有恢复——数据，这是不容易被训练过程精确控制的。也就是说，首次训练时第 5001 轮迭代训练的数据和现在“断点训练”的数据是不一样的。但是一般来说，只要保证每个训练批次（batch）内数据的分布相近，不会有太大的差异，两种训练都可以朝着正确的方向前进，其中存在的微小差距可以忽略不计。

第二种“再训练”的方式则是有理论基础支撑的训练模式。这个模式会在之前训练的基础上，对模型结构做一定的修改，然后应用到其他模型中。这种学习方式被称作**迁移学习**（Transfer Learning），这部分内容将在第 7 章介绍。这里举一个简单的例子，在当前模型训练完成之后，模型参数将被直接赋值到一个新的模型上，然后让这个新模型重头开始训练。这个操作可以通过下面这个命令完成：

```
./build/tools/caffe train -solver examples/mnist/lenet_solver.prototxt -
weights examples/mnist/lenet_iter_10000.caffemodel
```

执行命令后 Caffe 会像往常一样开始训练并输出大量日志，但是在完成初始化之后，它会输出这样一条日志：

```
I caffe.cpp:129] Finetuning from examples/mnist/lenet_iter_10000.
caffemodel
```

这条日志就是在告诉我们，当前的训练是在这个路径下的模型上进行“Finetune”。

### 5.1.4 训练日志分析

训练过程中 Caffe 产生了大量的日志，这些日志包含很多训练过程的信息，非常值得分析。分析的内容有很多，其中之一就是分析训练过程中目标函数 Loss 的变化曲线。在这个例子中，可以分析随着迭代轮数不断增加 Softmax Loss 的变化情况。首先将训练过程的日志信息保存下来，比方说日志信息被保存到 mnist.log 文件中，然后用下面的命令将 Iteration 和 Loss 的信息提取并保存下来：

```
grep Iteration mnist.log | grep loss | awk '{print $6,$9}' | sed 's/\,/'
> loss.data
```

提取后的信息可以用另一个脚本完成 Loss 曲线的绘图工作：

```
import matplotlib.pyplot as plt
x = []
y = []
with open('loss_data') as f:
    for line in f:
        sps = line[:-1].split()
        x.append(int(sps[0]))
        y.append(float(sps[1]))
plt.plot(x,y)
plt.show()
```

结果如图 5-1 所示，可见 Loss 很快就降到了很低的地方，模型的训练速度很快。这个优异的表现可以说明很多问题，这里就不做过多地分析了。

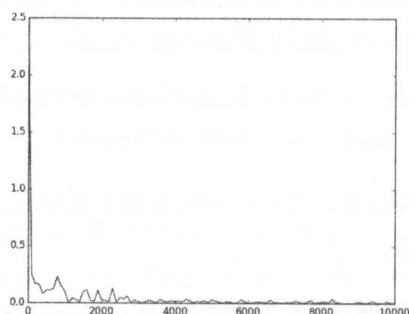


图 5-1 MNIST 数据训练的 Loss 曲线展示

除此之外，日志中输出的其他信息也可以被观察分析，比方说测试环节的精确度等，它们也可以通过上面的方法解析出来。由于采用的方法基本相同，这里就不赘述了，读者可以自行尝试。

正常训练过程中，日志里只会显示每一组迭代后模型训练的整体信息，如果想了解更多详细信息，就要将 solver.prototxt 中的调试信息打开，这样就可以获得更多有用的信息供读者分析：

```
debug_info:true
```

调试信息打开后，每一组迭代后每一层网络的前向后向计算过程中的详细信息都可以被观测到。这里截取其中一组迭代后的日志信息展示出来：

```
I sgd_solver.cpp:115] Iteration 0, lr = 0.01
I net.cpp:601]      [Forward] Layer mnist, top blob data data: 0.136623
I net.cpp:601]      [Forward] Layer mnist, top blob label data: 3.75
I net.cpp:601]      [Forward] Layer conv1, top blob conv1 data: 0.305875
I net.cpp:613]      [Forward] Layer conv1, param blob 0 data: 0.181242
I net.cpp:613]      [Forward] Layer conv1, param blob 1 data: 0.0818179
I net.cpp:601]      [Forward] Layer pool1, top blob pool1 data: 0.336549
I net.cpp:601]      [Forward] Layer conv2, top blob conv2 data: 0.881402
I net.cpp:613]      [Forward] Layer conv2, param blob 0 data: 0.0395974
I net.cpp:613]      [Forward] Layer conv2, param blob 1 data: 0.0192326
I net.cpp:601]      [Forward] Layer pool2, top blob pool2 data: 0.755979
I net.cpp:601]      [Forward] Layer ip1, top blob ip1 data: 0.925174
I net.cpp:613]      [Forward] Layer ip1, param blob 0 data: 0.0305547
I net.cpp:613]      [Forward] Layer ip1, param blob 1 data: 0.00411894
I net.cpp:601]      [Forward] Layer relu1, top blob ip1 data: 0.480685
I net.cpp:601]      [Forward] Layer ip2, top blob ip2 data: 2.83781
I net.cpp:613]      [Forward] Layer ip2, param blob 0 data: 0.044347
I net.cpp:613]      [Forward] Layer ip2, param blob 1 data: 0.0258073
I net.cpp:601]      [Forward] Layer loss, top blob loss data: 0.260246
I net.cpp:629]      [Backward] Layer loss, bottom blob ip2 diff:
0.000366457
I net.cpp:629]      [Backward] Layer ip2, bottom blob ip1 diff: 0.000101381
I net.cpp:640]      [Backward] Layer ip2, param blob 0 diff: 0.00720999
I net.cpp:640]      [Backward] Layer ip2, param blob 1 diff: 0.0130371
I net.cpp:629]      [Backward] Layer relu1, bottom blob ip1 diff: 5.37896e
-05
I net.cpp:629]      [Backward] Layer ip1, bottom blob pool2 diff: 7.39572e
-05
I net.cpp:640]      [Backward] Layer ip1, param blob 0 diff: 0.00121892
I net.cpp:640]      [Backward] Layer ip1, param blob 1 diff: 0.00138645
```

```
I net.cpp:629] [Backward] Layer pool2, bottom blob conv2 diff: 1.84893e-05
I net.cpp:629] [Backward] Layer conv2, bottom blob pool1 diff: 4.3865e-05
I net.cpp:640] [Backward] Layer conv2, param blob 0 diff: 0.00421591
I net.cpp:640] [Backward] Layer conv2, param blob 1 diff: 0.00695106
I net.cpp:629] [Backward] Layer pool1, bottom blob conv1 diff: 1.09662e-05
I net.cpp:640] [Backward] Layer conv1, param blob 0 diff: 0.00899409
I net.cpp:640] [Backward] Layer conv1, param blob 1 diff: 0.0206778
E net.cpp:729] [Backward] All net params (data, diff): L1 norm = (13529.1, 635.099); L2 norm = (24.2741, 1.79043)
```

想了解网络的更多表现，分析这些内容必不可少。

### 5.1.5 预测检验与分析

模型完成训练后，就要对它的训练表现做验证，看看它在其他测试数据集上的正确性。Caffe 提供了另外一个功能用于输出测试的结果。以下就是它的脚本：

```
./build/tools/caffe test --model=examples/mnist/lenet_train_test.prototxt
--weights=examples/mnist/lenet_iter_10000.caffemodel
```

脚本的输出结果如下所示：

```
I caffe.cpp:276] Batch 45, accuracy = 0.99
I caffe.cpp:276] Batch 45, loss = 0.0408834
I caffe.cpp:276] Batch 46, accuracy = 1
I caffe.cpp:276] Batch 46, loss = 0.0107022
I caffe.cpp:276] Batch 47, accuracy = 0.99
I caffe.cpp:276] Batch 47, loss = 0.0360289
I caffe.cpp:276] Batch 48, accuracy = 0.98
I caffe.cpp:276] Batch 48, loss = 0.0705472
I caffe.cpp:276] Batch 49, accuracy = 1
I caffe.cpp:276] Batch 49, loss = 0.00324827
I caffe.cpp:281] Loss: 0.0405774
I caffe.cpp:293] accuracy = 0.9882
I caffe.cpp:293] loss = 0.0405774 (* 1 = 0.0405774 loss)
```

除了完成测试的验证，有时读者还需要知道模型更多的运算细节，这就需要深入模型内部观察模型产生的中间结果。使用 Caffe 提供的接口，每一层网络输出的中间结果都可以用可视化的方法显示出来供大家观测、分析模型每一层的作用。其中的代码如下所示：

```
import numpy as np
import sys
import caffe
from skimage import io

def vis_square(data):
    # 这段代码是Caffe官方完成的例子
    data = (data - data.min()) / (data.max() - data.min())
    n = int(np.ceil(np.sqrt(data.shape[0])))
    padding = (((0, n ** 2 - data.shape[0]), (0, 1), (0, 1)) +
                ((0, 0),) * (data.ndim - 3))
    data = np.pad(data, padding, mode='constant', constant_values=1)

    data = data.reshape((n, n) + data.shape[1:]).transpose((0, 2, 1, 3) +
                    tuple(range(4, data.ndim + 1)))
    data = data.reshape((n * data.shape[1], n * data.shape[3]) + data.
                        shape[4:])
    return data

def predict(net, transformer, img):
    input_data = np.array(img)
    input_data = input_data.reshape(1, 28, 28, 1)
    net.blobs['data'].data[...] = transformer.preprocess('data', input_data
    [0])
    out = net.forward()

def process(model_path, weight_path, img_path):
    net = caffe.Net(model_path, weight_path, caffe.TEST)
    transformer = caffe.io.Transformer({'data':net.blobs['data'].data.
    shape})
    transformer.set_transpose('data', (2, 0, 1))

    img = caffe.io.load_image(img_path, color=False)
```

```
predict(net, transformer, img)
for key in net.blobs:
    data = net.blobs[key].data
    # 只可视化全连接层以上的结果
    if data.ndim == 4:
        vis = vis_square(data[0])
        io.imsave(key + '.png', vis)

if __name__ == '__main__':
    # 这里跳过参数校验
    model_path = sys.argv[1]
    weight_path = sys.argv[2]
    img_path = sys.argv[3]
    process(model_path, weight_path, img_path)
```

执行上面的代码就可以生成如图 5-2~ 图 5-5 所示的图像，它们各代表一个模型层的输出图像。

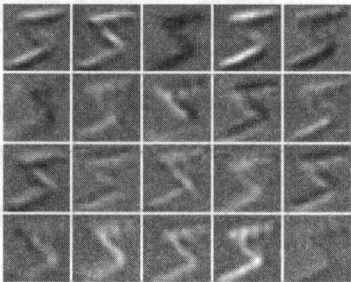


图 5-2 conv1 的输出图像

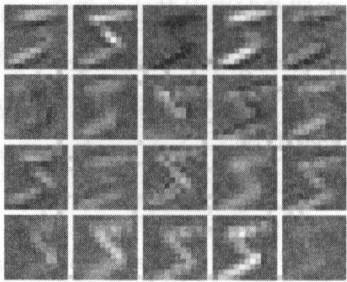


图 5-3 pool1 的输出图像

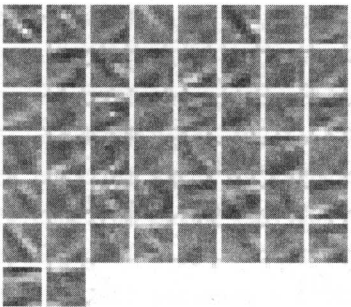


图 5-4 conv2 的输出图像

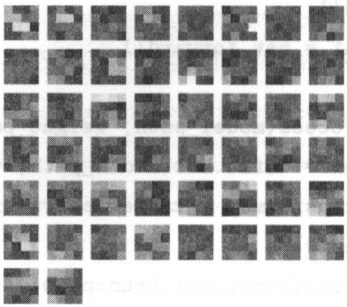


图 5-5 pool2 的输出图像

这组图展示了卷积神经网络是如何把一个数字转变成特征编码的。这样的方法虽然可以很好地看到模型内部的表现，例如 conv1 的结果图中有的提取了数字的边界，有的明确了前景像素所在的位置，这个现象和第 3 章中举例的卷积效果有几分相似。但是到了 conv2 的结果图中，模型的输出就变得让人有些看不懂了。实际上，想要真正看懂这些图像想表达的内容确实有些困难。

### 5.1.6 性能测试

除了在测试数据上的准确率，模型的运行时间也非常值得关心。如果模型的运行时间太长，甚至到了不可用的程度，那么即使它的精度很高也没有实际意义。测试时间的脚本如下所示：

```
./build/tools/caffe time -model examples/mnist/lenet_train_test.prototxt
```

Caffe 会正常完成前向后向的计算，并记录其中的时间。以下是一次测试结果的时间记录：

Average time per layer:

```
mnist    forward: 0.0562278 ms.
mnist    backward: 0.00204608 ms.
conv1    forward: 0.107064 ms.
conv1    backward: 0.22794 ms.
pool1    forward: 0.0385485 ms.
pool1    backward: 0.125717 ms.
conv2    forward: 0.190721 ms.
conv2    backward: 0.658197 ms.
pool2    forward: 0.0222291 ms.
pool2    backward: 0.052439 ms.
ip1      forward: 0.161355 ms.
ip1      backward: 0.132269 ms.
relu1    forward: 0.019143 ms.
relu1    backward: 0.0177619 ms.
ip2      forward: 0.0723578 ms.
ip2      backward: 0.0513517 ms.
loss     forward: 0.151423 ms.
loss     backward: 0.0309248 ms.
```

Average Forward pass: 0.921403 ms.

Average Backward pass: 1.39534 ms.



Average Forward-Backward: 2.42267 ms.

Total Time: 121.134 ms.

可以看出在性能测试的过程中，LeNet 模型只需要不到 1 毫秒的时间就可以完成前向计算，这个速度还是很快的。当然，这是在一个相对不错的 GPU 上运行的，那么如果在一个条件差的 GPU 上运行，结果如何呢？

Average time per layer:

mnist	forward:	1.70072 ms.
mnist	backward:	0.00189888 ms.
conv1	forward:	0.0826432 ms.
conv1	backward:	0.197758 ms.
pool1	forward:	0.0299686 ms.
pool1	backward:	0.114808 ms.
conv2	forward:	0.165917 ms.
conv2	backward:	0.598275 ms.
pool2	forward:	0.0162458 ms.
pool2	backward:	0.0460934 ms.
ip1	forward:	1.08853 ms.
ip1	backward:	0.127939 ms.
relu1	forward:	0.0114227 ms.
relu1	backward:	0.0159482 ms.
ip2	forward:	0.892079 ms.
ip2	backward:	0.0466797 ms.
loss	forward:	2.21752 ms.
loss	backward:	0.737255 ms.

Average Forward pass: 13.4363 ms.

Average Backward pass: 9.62471 ms.

Average Forward-Backward: 24.9117 ms.

Total Time: 1245.59 ms.

可以看到不同的环境对于模型运行的时间影响很大。

以上就是模型训练的一个完整过程。现在相信读者对深度学习模型的训练和使用有了基本了解。实际上，看到这里读者甚至可以扔下书亲自实践不同模型的效果，开始深度学习的实战之旅。

## 5.2 模型配置文件介绍

深度学习中一个十分重要的流程就是设计网络结构，本节将重点介绍 Caffe 中模型结构的配置方法。5.1 节已经介绍了模型的配置全部写在网络配置文件中，这个文件包含了所有网络层的配置信息，每个网络层的配置都被单独设置。下面就来详细介绍卷积神经网络中常见的一些模型层及它们在网络中的常见使用方法。

首先是卷积层，以下是卷积层的一种基本配置，并附上对应的注释。大多数的参数配置与此类似：

```
layer {
  name: "conv1" # 模型层的名称
  type: "Convolution" # 模型层的类型，这个有限制
  bottom: "data" # 层的输入数据名称
  top: "conv1" # 层的输出数据名称
  param { # 第一个参数的相关配置
    lr_mult: 1 # 参数的独立learning rate
  }
  param { # 第二个参数的相关配置
    lr_mult: 2
  }
  convolution_param { # 卷积层自身的配置
    num_output: 20 # 输出的channel数量
    kernel_size: 5 # 卷积核的维度: 5 * 5
    stride: 1 # 卷积核计算的跨度: 1
    pad: 0 # 卷积操作的填边: 0
    weight_filler { # 卷积参数的初始化: xavier(后面会说)
      type: "xavier"
    }
    bias_filler { # 偏置项的初始化: 初始为常数0
      type: "constant"
    }
  }
}
```

可以看出模型层的配置分成两个部分：层的通用部分和层的独立部分。通用部分主要设置最基本的配置，比方说层的输入输出，以及层内参数的一些基础配置（上面配置中的参数学习率）。独立部分就是具体某个类别的模型层独有的配置。与卷积层相比，这里有输出的 channel 数量等，参数的含义在第 4 章已经介绍过。

下面是全连接层的配置，和卷积层相比，全连接层的内容就显得简单了许多：

```
layer { # 前面的内容不需要介绍了
  name: "ip1"
  type: "InnerProduct"
  bottom: "pool2"
  top: "ip1"
  param {
    lr_mult: 1
  }
  param {
    lr_mult: 2
  }
  inner_product_param { # 全连接层的参数
    num_output: 500 # 输出的维度
    weight_filler { # 权重项的初始化
      type: "xavier"
    }
    bias_filler { # 偏置项的初始化
      type: "constant"
    }
  }
}
```

看完了两个最核心的模型层，再看看其他的层。接下来一个常见的层就是全连接层和卷积层中的非线性部分，在 Caffe 中线性和非线性这两个部分被拆成了两个独立的模型层，非线性部分的基本配置如下所示，如果想更换其他种类的非线性函数，可以修改 type 对应的配置值：

```
layer {
  name: "relu1"
  type: "ReLU"
  bottom: "ip1"
  top: "ip1"
}
```

另外一个在图像处理中十分重要的层是 Pooling 层，它的中文翻译比较晦涩，实际上可以把它想象成一个聚合并丢弃非重要信息的模型层，它的操作方式类似卷积操作，选定 feature map 中的一个局部区域，对这些区域进行某个简单的操作，例如取最大

值或者求平均值。一般来说，经过 Pooling 层的计算，图像的 feature map 会变小，但是其中的信息量并没有明显减少。另外，Pooling 层的操作虽然简单，但实际上它是两个操作的结合，正如这个模型层的全名一样。Max Pooling 是 Max 操作和 Pooling 操作两个操作的结合。所谓的 Max 操作和一些图像处理中的非极大抑制（Non Max Suppression）思想一致，而 Pooling 和图像处理中的尺度缩放思想一致。Average 操作的语意和 Max 操作的语意又有所不同，这些差别需要认真分析考虑。下面是它的配置，内容比较简单：

```
layer {
  name: "pool2"
  type: "Pooling"
  bottom: "conv2"
  top: "pool2"
  # 这表示pooling层是MAX Pooling，每次选择2*2的区域中最大的一个作为代表输出
  pooling_param {
    pool: MAX
    kernel_size: 2
    stride: 2
  }
}
```

看完了几个模型中的关键计算部件，接下来看看模型的 IO 部分。首先是训练阶段模型的输入数据层，这个模型层的内容比较多，一般会包含模型的读入配置和数据预处理的配置。它的配置如下所示：

```
layer {
  name: "mnist"
  type: "Data"
  top: "data" # 一般的数据层会输出这两部分
  top: "label"
  include { # 这个配置主要标明模型层所涉及的阶段：TRAIN还是TEST
    phase: TRAIN # 设定这个数据只用于训练阶段
  }
  transform_param {# 对输入数据做初始化
    scale: 0.00390625 # 所有数据做归一化：这个数字是1/255的近似值
  }
  data_param { # 数据参数
    source: "examples/mnist/mnist_train_lmdb" # DB路径
    batch_size: 64 # 一次训练的图片量
  }
}
```

```
    backend: LMDB # 告诉caffe数据库的格式
  }
}
```

训练阶段模型的输出就是 Loss 层，这部分的模式相对固定，配置比较简单：

```
layer {
  name: "loss"
  type: "SoftmaxWithLoss"
  bottom: "ip2"
  bottom: "label"
  top: "loss"
}
```

与训练阶段的 IO 不同，测试阶段模型的数据输入层一般会有些不同，看上去更简单：

```
layer {
  name: "data"
  type: "Input"
  top: "data"
  input_param { shape: { dim: 1 dim: 3 dim: 32 dim: 32 } }
}
```

测试阶段模型的数据输出层也是如此，不需要再去计算 Loss，只计算 Softmax 的结果就好：

```
layer {
  name: "prob"
  type: "Softmax"
  bottom: "ip2"
  top: "prob"
}
```

只要能看懂上面的这些模型结构，绝大多数的模型就可以轻松应付了。至于那些拥有独特结构的模型，我们遇到它们时再做分析。下面的内容十分重要，如果想更全面地了解每一个模型层拥有的配置该怎么做呢？这时可以查看模型层配置的定义文件：*src/caffe/proto/caffe.proto*。这个文件按照 Protobuf 的格式定义了网络结构和求解所需的各种定义。最终也会利用 Protobuf 生成对应的源文件。例如，对于其中的 Convolution-Parameter，它的定义如下所示：

```

message ConvolutionParameter {
  optional uint32 num_output = 1; // The number of outputs for the layer
  optional bool bias_term = 2 [default = true]; // whether to have bias
  terms

  // Pad, kernel size, and stride are all given as a single value for
  equal
  // dimensions in all spatial dimensions, or once per spatial dimension.
  repeated uint32 pad = 3; // The padding size; defaults to 0
  repeated uint32 kernel_size = 4; // The kernel size
  repeated uint32 stride = 6; // The stride; defaults to 1
  // Factor used to dilate the kernel, (implicitly) zero-filling the
  resulting
  // holes. (Kernel dilation is sometimes referred to by its use in the
  // algorithme à trous from Holschneider et al. 1987.)
  repeated uint32 dilation = 18; // The dilation; defaults to 1

  // For 2D convolution only, the *_h and *_w versions may also be used to
  // specify both spatial dimensions.
  optional uint32 pad_h = 9 [default = 0]; // The padding height (2D only)
  optional uint32 pad_w = 10 [default = 0]; // The padding width (2D only)
  optional uint32 kernel_h = 11; // The kernel height (2D only)
  optional uint32 kernel_w = 12; // The kernel width (2D only)
  optional uint32 stride_h = 13; // The stride height (2D only)
  optional uint32 stride_w = 14; // The stride width (2D only)

  optional uint32 group = 5 [default = 1]; // The group size for group
  conv

  optional FillerParameter weight_filler = 7; // The filler for the weight
  optional FillerParameter bias_filler = 8; // The filler for the bias
  enum Engine {
    DEFAULT = 0;
    CAFFE = 1;
    CUDNN = 2;
  }
  optional Engine engine = 15 [default = DEFAULT];
}

```

```

// The axis to interpret as "channels" when performing convolution.
// Preceding dimensions are treated as independent inputs;
// succeeding dimensions are treated as "spatial".
// With (N, C, H, W) inputs, and axis == 1 (the default), we perform
// N independent 2D convolutions, sliding C-channel (or (C/g)-channels,
for
// groups g>1) filters across the spatial axes (H, W) of the input.
// With (N, C, D, H, W) inputs, and axis == 1, we perform
// N independent 3D convolutions, sliding (C/g)-channels
// filters across the spatial axes (D, H, W) of the input.
optional int32 axis = 16 [default = 1];

// Whether to force use of the general ND convolution, even if a
specific
// implementation for blobs of the appropriate number of spatial
dimensions
// is available. (Currently, there is only a 2D-specific convolution
// implementation; for input blobs with num_axes != 2, this option is
// ignored and the ND implementation will be used.)
optional bool force_nd_im2col = 17 [default = false];
}

```

一般来说，在定义网络时不会使用全部的配置，只会用到其中最常用的一部分，但是如果想做更多、更精细的设置，充分了解其中的内容还是很有必要的。以上就是Caffe使用的一些常用知识，拥有了这些知识就可以做下面更有挑战但也更有意义的事情——观察Caffe的整体结构，阅读Caffe的源码。

## 5.3 Caffe 的整体结构

本章的目标之一是深入剖析一个成熟的深度学习框架，看看它的源代码架构及各部分的实现方式，使读者对深度学习框架有一个初步的认识。

只要思路正确，阅读代码并不是一件很困难的事情。不过在阅读代码之前读者还要先回答两个问题。

1. 阅读代码是为了什么？没有目标的阅读会让我们的努力事倍功半。
2. 代码阅读到什么程度呢？这个问题实际上和上一个问题相关，还是为目标服务。

一般来说, 阅读代码有下面几个目的。

1. 搞清楚代码实现的算法或者功能的基本情况。这个目的适合对算法本身不是很了解, 希望通过阅读代码了解算法的人。
2. 搞清楚代码在实现过程中的细节。这种情况下, 一般对算法已经有大概的了解, 读代码是为了了解代码中对算法细节的考量。当然, 如果想要精通算法, 了解代码细节是很有帮助的。
3. 扩展源代码。在开源代码的基础上, 利用已有的框架, 增加或者修改功能来实现自己想要的功能。首先需要对代码的架构细节有深入的了解, 其次对一些关键算法要清楚。

这里读者的首要目标是学会如何扩展代码。模型和相应的技术在不断发展, 如果想要跟上前进的步伐, 那么学会扩展代码就变得非常有意义。Caffe 中主要的扩展点就是 Layer, 也就是神经网络的网络层模块, 新的网络模型需要新的网络层支持。当然, 其他的模块也可以扩展, 比方说求解算法模块 Solver, 数据读入模块 Data Reader。

确定了上面第一个问题, 下面就要解决第二个问题。读代码要读到什么程度? 一般来说, 阅读代码这件事情可以用一个 Logistic 型的函数表示, 如图 5-6 所示。

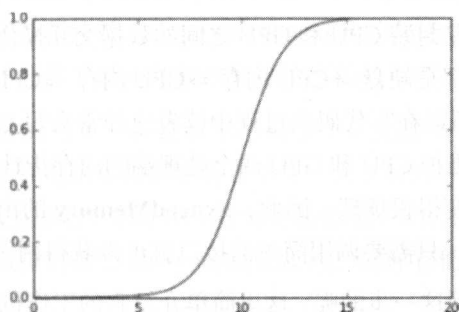


图 5-6 阅读源代码花费的时间和从中得到的收获之间的关系, 其中横轴表示花费的时间, 纵轴表示收获。由于不同的项目有各自的特点, 所以上面的曲线只作概念上的展示, 不做量化分析

横轴是阅读代码花费的时间, 纵轴是阅读代码带来的效果。对于代码量比较大的项目, 刚开始读肯定很困惑, 大家需要花很长的时间梳理清楚各个文件、各个模块之间的关系, 花在这些事情上的时间非常难熬; 随着结构关系逐渐清晰, 大家开始领会代码中各个模块的关系和项目的基本架构, 阅读代码的效果直线上升, 阅读的快乐也逐渐体现出来; 当把代码主线和重要支线看懂后, 再读一些小支线和一些复杂的模块时, 难度又会直线上升, 而且收益也不会太大, 痛苦会又一次笼罩着大家。所以根据阅读代码的性价比和 Caffe 代码自身的特点, 本章只会将代码主线和一些重要支线介绍完, 预



计可以涵盖整体代码量的一半。理解了本章介绍的内容后，相信读者可以应付 Caffe 中绝大多数的问题，这时如果想去剩下的代码也不会感到困难重重。

不同于其他面向深度学习的框架，Caffe 没有采用符号计算的模式编写，整体上的架构由以下几个核心的模块类组成。

- SyncedMemory

- Blob

- Layer

- Net

- Solver

- 多 GPU 训练

- IO

下面就来简单介绍各部分的内容。

### 5.3.1 SyncedMemory

这个类的主要功能是封装 CPU 和 GPU 之间的数据交互操作。一般来说，模型训练预测时数据的流动形式都是硬盘  $\Rightarrow$  CPU 内存  $\Rightarrow$  GPU 内存  $\Rightarrow$  CPU 内存  $\Rightarrow$  GPU 内存  $\cdots \cdots \Rightarrow$  CPU 内存  $\Rightarrow$  硬盘，所以在写代码的过程中读者也经常会写 CPU/GPU 之间数据传输的代码，同时还要分别维护 CPU 和 GPU 两个处理端的内存指针。这些事情处理起来不会很难，但是会使代码变得很烦琐。因此，SyncedMemory 的出现就是把 CPU/GPU 的数据传输操作封装起来，只需要调用简单的接口就可以获得两个处理端同步后的数据。

那么，它是做到这一步的呢？这里简单介绍它的工作机理，首先给出它的核心数据结构：

```
class SyncedMemory {
    void* cpu_ptr_;
    void* gpu_ptr_;
    SyncedHead head_;
}

enum SyncedHead { UNINITIALIZED, HEAD_AT_CPU, HEAD_AT_GPU, SYNCED };
```

其中的 head\_ 保存了数据的状态，任何时刻总有某一个端（CPU 或 GPU）上的数据是最新的，那么当我们去取另一个端上的数据时，SyncedMemory 就会自动和最新的数据同步，其中的逻辑可以用一个精简过的代码表示：

```
inline void SyncedMemory::to_gpu() {
    switch (head_) {
        case HEAD_AT_CPU:
            caffe_gpu_memcpy(size_, cpu_ptr_, gpu_ptr_);
            head_ = SYNCED;
            break;
    }
}
```

当数据处于 HEAD\_AT\_CPU，也就是说 CPU 的数据处于最新状态时，如果想取得 GPU 数据，就要让 GPU 数据和 CPU 数据进行同步。这时 CPU 和 GPU 的数据已经同步，状态就可以更改为 SYNCED。

### 5.3.2 Blob

**Blob** 这个类在 SyncedMemory 的基础上做了两个封装。

1. 操作数据的封装。Caffe 中的 Blob 实际上是一个 4 维的张量 ( $N, C, H, W$ )，张量就是另一个知名开源软件 TensorFlow 中 tensor 的翻译。Blob 提供了许多数据访问的接口，同时还提供了访问和修改数据维度的接口等。
2. 对数据和其更新量的封装。在机器学习模型中优化模型参数是一件必做的事情，所以参数的原始量和更新量都需要保存下来。为了处理方便，每一个 Blob 中都有 data 和 diff 两个数据指针，data 用于存储原始数据，diff 用于存储反向传播的梯度更新值。这两个数据的数据原型是 SyncedMemory，这样它们也拥有不同处理端访问的便利。

就这样 Blob 基本完成了整个 Caffe 数据部分结构的封装，在 Net 类中我们可以看到，所有前后向计算的数据和参数相关的需求都可以由 Blob 实现。

### 5.3.3 Layer

完成了数据层次的抽象，接下来就是模型层次的抽象。在前面的章节中曾经讲过，神经网络的前后向计算可以做到层与层之间完全独立，那么每个层只要依照一定的接口规则实现，将需要的参数传递给前后层，整个网络的正确性就可以得到保证。Caffe 实现了一个基础的抽象类 **Layer** 用于表示模型层，并定义了许多模型层创建和运算必需的接口。所有实现具体模型层功能的类都要继承它并实现其中定义的抽象方法，对

于一些复杂功能的模型层还会有自己的抽象类（比如卷积层的 `base_conv_layer`）。Layer 类主要采用了模板的设计模式（Template）方法设计，基类里实现了一些基本功能的操作流程，子类里实现具体的操作。每一个子类都通过工厂方法创建，子类的具体方法通过多态的方式调用得到。这样当读者需要实现一个新的模型层时，琐碎的事情不再需要处理，只需要关心模型层的初始化和前后向计算即可。这部分内容将在 5.4 节中详细介绍。

### 5.3.4 Net

**Net** 作为网络模型的表示类，将数据和模型层的信息组合起来做进一步封装。它对外只暴露模型的初始化和前后向计算的接口，使得整体看上去和一个模型层的接口类似，但同时还把复杂的逻辑计算全部封装起来。除此之外，Net 保存了以下信息。

1. 每一个模型层的输入输出数据。
2. 每一个模型层参数的指针，用来索引模型参数信息。为了实现更复杂的模型，不同的模型层之间还可以实现参数共享（5.5 节会有详细介绍）。

Net 通过读取模型配置文件进行初始化，并明确模型中的内容，从而实现不同种类的网络模型。这部分内容将在 5.5 节中详细介绍。

### 5.3.5 Solver

拥有了 Net 和模型结构，实际上读者已经可以进行网络的前向后向计算了，模型还缺乏有关网络的学习训练的功能。于是在此基础上，**Solver** 类进一步封装了一些训练和预测相关的功能。与此同时，它还开放了两类方法。

一类是计算参数更新量的方法。在 Solver 类中这是一个抽象方法，它的输入包含了反向计算得到的梯度，输出则是模型最终的更新量。想要提供给 Caffe 新的优化方法，只要继承 Solver 类并实现相应的参数更新方法即可。Caffe 已经准备了很多如大家常见的优化方法，比如 Momentum、Adagrad、Adam 等，只需填入相应的参数就可以使用这些方法。

另外一类是训练过程中注入的回调函数。用户可以向 Solver 注入一些回调函数，这个函数会在模型每一轮训练前和反向计算后被调用。如果对优化过程有特殊的需求，不需要重写优化的流程，只需要注入实现目标功能的回调函数即可。Caffe 代码中就有使用回调函数的例子。例子中回调函数被用于实现单机多 GPU 的并发训练算法。

Solver 的内容将在 5.6 节详细介绍。

5.3.6 多 GPU 训练

对于单 GPU 的训练和预测来说，Caffe 的基本层次关系到这里就结束了，如果要进行多 GPU 训练，上层还需要 **InternalThread** 和 **P2PSync** 两个类，用以实现基于数据并行的并发优化算法。这部分内容会在 5.6 节中详细描述。

5.3.7 IO

模型训练还需要输入训练数据的模块和初始化参数的模块，正所谓巧妇难为无米之炊，没有数据一切都是空谈。**Data Reader** 和 **Data Transformer** 可以帮助读者输入数据并完成数据预处理，**Filler** 可以对参数进行定制初始化，还有一些操作 **Snapshot** 的方法帮助模型完成持久化工作，这样模型和数据的 IO 问题也就解决了。这部分的内容将在 5.7 节详细描述。

到此为止，Caffe 的主线功能就介绍完了。这里可以用一张大图把 Caffe 的整体层次关系和核心部件展示出来，如图 5-7 所示。

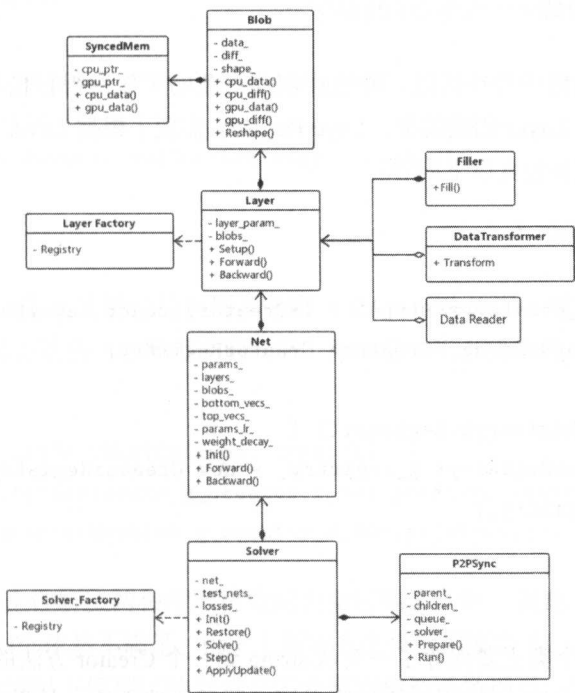


图 5-7 Caffe 的核心架构图

除了展示了上面提到的模块之间的关系，还列出了每个模块类中的核心属性和方法。如果读者已经对图 5-7 中展示的内容和图中的一些细节比较清楚，那么对 Caffe 的了解应该已经非常不错了，本章后面的内容甚至可以跳过不看；如果还有不清楚的地方，就请继续往下看。

## 5.4 Caffe 的 Layer

本节主要介绍 Caffe 的 Layer 模块。其中重点关注 Layer 的创建、初始化和前向后向计算三个部分。作为深度学习最灵活的模块，网络结构需要能被配置生成，易用性和易扩展性都需要被考虑进去。为了做到这一点，Caffe 主要实现了两个类。

- LayerRegistry：辅助实现 Layer 的动态创建工作。
- Layer：实现模型层的初始化和计算流程的接口，方便模型层实现自己的功能。

下面我们将一一介绍。

### 5.4.1 Layer 的创建——LayerRegistry

设计模式的书籍中曾经讲过，要想让创建对象的工作变得轻松灵活，工厂方法必不可少。作为管理 Layer 创建的类，LayerRegistry 实现了根据 Layer 名字创建 Layer 类的工作。它里面主要包含以下内容：

```
class LayerRegistry {
public:
    typedef shared_ptr<Layer<Dtype> > (*Creator)(const LayerParameter&);
    typedef std::map<string, Creator> CreatorRegistry;

    static CreatorRegistry& Registry() {
        static CreatorRegistry* g_registry_ = new CreatorRegistry();
        return *g_registry_;
    }
}
```

可以看出，这个类主要保存了一个从 string 到一个 Creator 方法的映射。其中 string 就是一个模型层的名字，Creator 就是这个模型层的构造方法。如果想根据一个名字创建一个 Layer，可以调用下面的代码完成（基于源代码做了一定简化）：

```

shared_ptr<Layer<Dtype>> CreateLayer(const LayerParameter& param) {
    const string& type = param.type();
    CreatorRegistry& registry = Registry();
    return registry[type](param);
}

```

有了这个方法后，最核心的“工厂方法”设计模式就被构建起来了。当然，下面又有一个实际的问题出现了。CreatorRegistry 属性保存了这个映射，那么如何把模型层名字和构造方法的映射写入这个 map 中呢？完成这件事情又需要点小技巧。

LayerRegistry 提供了一个方法，这个方法可以把这些配对信息注册到它的映射中：

```

static void AddCreator(const string& type, Creator creator) {
    CreatorRegistry& registry = Registry();
    registry[type] = creator;
}

```

有了注册的方法，Caffe 还设计了另一个类 LayerRegisterer，这个类的构造函数将完成一个由类名到创建方法映射的添加：

```

class LayerRegisterer {
public:
    LayerRegisterer(string& type, shared_ptr<Layer<Dtype>> > (*creator)(const
        LayerParameter&)) {
        LayerRegistry<Dtype>::AddCreator(type, creator);
    }
};

```

有了这个类，只要在代码的某个角落声明一个 LayerRegisterer 类，并在类构造时把 Layer 的名字和构造方法传入，即可完成注册。为了进一步方便注册，Caffe 还提供了一个宏简化流程：

```

#define REGISTER_LAYER_CREATOR(type, creator) \
static LayerRegisterer<float> g_creator_f_##type(#type, creator<float>);\
static LayerRegisterer<double> g_creator_d_##type(#type, creator<double>);\

```

这个宏可以一次性解决两种浮点精度的模型层构造注册，到此注册的任务算是基本完成了，但是 Caffe 依然不满足。由于上面的方法依然需要定义一个创建模型层对象的方法，而大多数情况下模型层的创建只需要调用构造函数即可，因此注册过程在上面的基础上做了更进一步简化：

```

#define REGISTER_LAYER_CLASS(type) \
    template <typename Dtype> \
    shared_ptr<Layer<Dtype> > Creator_##type##Layer(const LayerParameter& \
    param) \
    { \
        return shared_ptr<Layer<Dtype> >(new type##Layer<Dtype>(param)); \
    } \
    REGISTER_LAYER_CREATOR(type, Creator_##type##Layer)
}

```

有了这个宏，只要约定好模型层的名字和对应类别的名字，并在类中实现标准的构造函数，就可以直接用这个方法完成注册。由于 C++ 不是一个足够动态的语言，编码实现类似其他语言中“反射”这样的功能还是有点复杂的。当读者去看一些具体的模型层代码时就会发现在它的 `cpp` 文件中有这个宏的调用，其目的就是完成动态创建的注册。总的来说，Caffe 采取解耦合的方式注册模型层类的构造方法，使得模型层类可以根据配置定义灵活创建。

讲完了类变量创建的灵活性，下面介绍构造参数的灵活性。上面的代码中还有一个十分关键的类：LayerParameter，这个类的定义同样保存在 `[Caffe Home]/src/caffe/proto/caffe.proto` 中，它将在编译时由 Protobuf 自动生成。由于不同的模型层相差很大，需要的参数也千差万别，因此这个文件记录了一个非常复杂的 LayerParameter 结构。由于 Protobuf 提供了一系列的基础功能——类的构造、成员变量设置、序列化等功能，这部分的操作同样保持了足够的灵活性，同时并不需要完成太多的代码编写。对这部分内容感兴趣的读者请自行阅读定义文件中的信息，这部分内容在 4.2 节已经有介绍，这里就不再赘述了。

上面提到的这些内容都是 `layer_factory.hpp` 中的内容，那么 `cpp` 中的内容呢？`cpp` 除了提供一些经典的 Layer 的构造方法之外，还实现了根据是否使用 cuDNN 库情况自动切换模型层的功能。cuDNN 库是 CUDA 平台下专门针对深度学习设计的运算库，利用这个库我们可以获得更快的运算性能。这里就不得不多说一句，在安装 Caffe 时，最好把这个库装上，这个库带来的速度提升是非常显著的。

LayerRegistry 的工作介绍完了，下一步就来关注创建和初始化模型层中的一些细节问题。

## 5.4.2 Layer 的初始化

下面开始介绍 Layer 的初始化过程。Layer 的构造函数并不是它初始化的核心函数，真正初始化的工作全在下面这个函数完成：

```

void SetUp(const vector<Blob<Dtype>*>& bottom,
           const vector<Blob<Dtype>*>& top) {
    InitMutex();
    CheckBlobCounts(bottom, top);
    LayerSetUp(bottom, top);
    Reshape(bottom, top);
    SetLossWeights(top);
}

```

这个函数调用了 5 个方法，但实际上后面 3 个方法更重要。下面就来依次看看它们。首先是 `LayerSetup` 和 `Reshape`。这两个方法在 `Layer` 类中都被定义为抽象方法，也就是说，它们的具体内容要靠子类实现。所以说这里的初始化方法采用了“模板”的设计模式，父类明确流程，子类负责实现细节。这两个方法从名字上看有不同的分工。

- `LayerSetup`：对模型层进行初始化。
- `Reshape`：根据 `bottom`（也就是层的输入数据）的维度，确定 `top`（也就是层的输出数据）的维度。

虽然这两个方法有不同的分工，但是由于传入的参数完全一样，而且都处于初始化环节，有些 `Layer` 的作者在实现自己特殊的 `Layer` 时会把这两个函数合二为一，只实现一个。实际上这样做也是可以的。

最后一个函数 `SetLossWeights` 在 `Layer` 中有具体实现，它的作用是帮助模型为训练过程中的前向后向计算做准备工作。`Caffe` 中的 `Loss Layer` 和其他 `Layer` 不同，`Loss` 要输出训练模型的最终结果——损失，同时还要将损失回传到反向计算中，可以说 `Loss Layer` 是模型前后向计算的转折点。为了确保模型计算能够顺利，`Loss Layer` 需要完成计算以外的额外工作，而这个函数就是为这些额外工作做准备。

顺着上面的内容，在配置文件中哪里可以设置一个模型层是一个 `Loss Layer` 呢？实际上 `Caffe` 单独实现了一个 `Loss Layer`，并在这个类的 `LayerSetup` 函数中为它的子类自动添加了模型参数的 `loss_weight`。相关代码如下所示：

```

void LossLayer<Dtype>::LayerSetUp(
    const vector<Blob<Dtype>*>& bottom, const vector<Blob<Dtype>*>& top) {
    // LossLayers have a non-zero (1) loss by default.
    if (this->layer_param_.loss_weight_size() == 0) {
        this->layer_param_.add_loss_weight(Dtype(1));
    }
}

```



这部分代码的大意是：如果在配置文件中没有明确设置 `loss_weight` 的值，就为这个类的参数加一个 `loss_weight`。这样所有继承了 `Loss Layer` 的模型层类只要调用这个方法就可以自动完成 `loss_weight` 的设置——即使在配置文件中不用显式定义。例如，大家经常使用的 `SoftmaxWithLossLayer`：

```
void SoftmaxWithLossLayer<Dtype>::LayerSetUp(
    const vector<Blob<Dtype>*>& bottom, const vector<Blob<Dtype>*>& top) {
    LossLayer<Dtype>::LayerSetUp(bottom, top);
    // 以下省略
}
```

通过调用父类的方法，`loss_weight` 参数自动设置完成。如果我们想实现一个新的 `Loss Layer`，那么一定要记得调用一个父类的这个方法设置 `loss_weight`。

如果参数 `loss_weight` 被设定，那么下面的代码就比较容易理解了（以下代码经过修改）：

```
for (int top_id = 0; top_id < top.size(); ++top_id) {
    const Dtype loss_weight = layer_param_.loss_weight(top_id);
    loss_[top_id] = loss_weight;
    const int count = top[top_id]->count();
    Dtype* loss_multiplier = top[top_id]->mutable_cpu_diff();
    caffe_set(count, loss_weight, loss_multiplier);
}
```

从代码中可以看出，`Layer` 输出的 `loss` 权重被设为 `loss_weight`，同时输出位置的梯度值也被预先设为 `loss_weight`。如果是默认设置，那么这两个地方的值就等于 1。这两处设置完之后会起什么样的作用呢？这部分的故事就要放在 `Layer` 的前向计算中介绍了。

### 5.4.3 Layer 的前向计算

实际上，模型层的前向后向计算同样采用了 `Template` 的设计模式进行封装，后向计算的封装相对简单，所以重点介绍前向计算的这部分代码：

```
Dtype loss = 0;
// 省略部分代码
case Caffe::CPU:
    Forward_cpu(bottom, top);
```

```

for (int top_id = 0; top_id < top.size(); ++top_id) {
    if (!loss_[top_id]) { continue; }
    const int count = top[top_id]->count();
    const Dtype* data = top[top_id]->cpu_data();
    const Dtype* loss_weights = top[top_id]->cpu_diff();
    loss += caffe_cpu_dot(count, data, loss_weights);
}
break;
// 省略部分代码
return loss;

```

Layer 前向计算的函数要输出网络最终的 Loss，所以在完成了前向计算后，需要统计 Loss 的总和。由于前面初始化时 Layer 把 loss\_weight 保存在了 top 数据的 cpu\_diff (或者 gpu\_diff)，现在我们就可以把这两部分乘起来得到最终值。当然，因为绝大多数的 loss\_weight 为 1，所以这里的结果其实就是 Loss 的加和，这样就清楚 Loss 计算的全部流程了。

以上就是模型层 Layer 在架构上的表现，我们对 Layer 的构造、初始化和前向后向计算有了一定的了解，这些内容将帮助读者了解下面的内容。

## 5.5 Caffe 的 Net 组装流程

前面提到 Net 类的主要功能有两个：模型组装和模型计算。其中模型计算的功能十分简单，它通过调用每一个 Layer 的前后向计算就完成了。而且在 5.4 节读者已经看到，Loss 计算相关的工作都是由 Layer 自己完成的，所以 Net 类中这部分代码基本没有加入新的内容。所以本节的重点将放在模型组装这个功能上。

为了更好地了解 Caffe 组装模型的过程，这一次将不去直接看 Net 组装功能的代码，而是通过阅读训练前组装模型的 log 来了解组装的全过程。这段 log 总体来说展示了 Train 阶段和 Test 阶段的两个网络组装的全过程。如果模型配置正确，这段 Log 在大家训练网络时一般都是一闪而过，无暇看清；如果它没有一闪而过而是停下来了，那么很有可能是读者的网络配置有问题。所以了解这段 log 的内容对于调试代码非常有用。

为了展示模型组装的全过程，本书在这里选取了一个实际例子，就是 Caffe 的 examples 里面的 Siamese Model。Siamese Model 是度量学习中的一个经典模型，所谓的度量学习是为了找到一种方法可以测量我们关注的事物之间的距离。这个世界上有许多真实存在的空间，也有许多存在于概念中的空间。有的空间存在一种度量方法，可以衡

量空间中每一对元素间的距离；有的空间则无法使用常规的度量方法度量，那就需要读者亲自找到一种方法度量元素之间的距离。例如 MNIST 数据集中的两个数字，我们可以用分类识别的方式知晓它的类别信息，但是我们怎么能知道每张图像之间的距离呢？如果这个距离表示两个数字图像的相似度，那么对于一个写着“8”的图像，它与写着“1”的图像近还是与写着“2”的图像近呢？还是一样近呢？某些场景下，回答这样的问题是十分必要的，因为只有定义了度量，空间才会具有更多的运算性质，更多的信息才能被挖掘出来。

机器学习中接触到的度量学习有连续型度量，也有离散型度量。这里的例子是离散型度量。度量学习中的标注数据和分类识别中的不同，它一般有三种类型的标注数据。

- 某个数据和另一个数据“相似”
- 某个数据和另一个数据“不同”
- 对于某个数据 A 来说，在数据 B 和数据 C 之间，它更“像”数据 A

这个例子只拥有前面两种训练数据。

模型的目标是输入两张图像并判断这两张图像是否相似。它是由两个完全相同的模型组成，经过网络计算后，输入的图片将各自变换成描述其特征的向量，最终的目标函数就是测量这两个向量在欧式空间的距离。如果两张图像应该被判为相似，那么两个向量最好能够相等；如果两张图像不相似，那么两个向量在欧式空间至少要保持一定的距离。这个模型可以用图 5-8 表示。

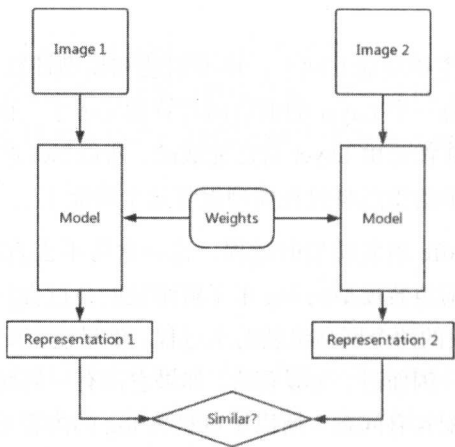


图 5-8 Siamese 网络结构图

这个网络除了包含其他正常网络中的一些特性之外，还具有网络参数复用的特点，这一点在后面的分析中将被特别提到。

介绍完了模型的背景，下面来重点关注其中几个具有代表性的片段。日志中的网络构建首先是一个正常的卷积层 conv1，Log 如下所示（以下代码的行号可能会有不同，但位置是相近的）：

```
layer_factory.hpp:77] Creating layer conv1
net.cpp:92] Creating Layer conv1
net.cpp:428] conv1 <- data
net.cpp:402] conv1 -> conv1
net.cpp:144] Setting up conv1
net.cpp:151] Top shape: 64 20 24 24 (737280)
net.cpp:159] Memory required for data: 3752192
```

Log 第一行是创建这个 Layer 实例的代码，具体的创建过程在 LayerRegistry 里面。这部分的原理我们在 5.4.1 节已经介绍，只要提供 Layer 的名字（配置文件中的参数名叫 type），就可以根据名字和对应参数实例化一个 Layer 类。

第 3 行和第 4 行显示了初始化 conv1 的 bottom 和 top 数据的过程。在完成了 Layer 的创建后，Net 就要准备好这个 Layer 的输入输出数据，以便进行 Layer 的初始化。这里涉及 Net 类中的 AppendBottom 和 AppendTop 两个方法，由于这两个方法中主要包含了一些数据结构的存储代码，比较烦琐且读懂的价值不算太大，所以在此不再详细介绍。因为每一个 bottom blob 和 top blob 都有自己的名字，所以日志中就将它们和 Layer 的关系输出出来。

第 5 行代表了 Layer 的 Setup 函数已经调用完成（或者这个 Layer 和其他 Layer 共享）。这部分内容在 4.4 节已经介绍过，而第 6 行在 Layer 初始化，明确了 top 数据维度后，输出 top 数据的维度。第 7 行则是统计模型累积到这里时，top 数据消耗的内存容量。可以看出截至到这一层，内存消耗大约是 3MB 多，还不算大。

上面的片段就是一个最典型的拥有参数的 Layer 初始化输出的日志，下面一段是构建 ReLU 层输出的日志，这段日志同前面一段相比稍有不同：

```
layer_factory.hpp:77] Creating layer relu1
net.cpp:92] Creating Layer relu1
net.cpp:428] relu1 <- ip1
net.cpp:389] relu1 -> ip1 (in-place)
net.cpp:144] Setting up relu1
net.cpp:151] Top shape: 64 500 (32000)
net.cpp:159] Memory required for data: 5769472
```

最不同的就是第 4 行结尾的 (in-place)，这说明 relu1 这一层的 bottom blob 和 top blob 是同一个数据，这和我们在网络中的定义是一样的——两个数据同名。in-place 的

好处就是节省内存，减少内存复制等数据操作，但奇怪的是这里在统计内存消耗时并没有考虑 in-place 带来的节省。

接下来的一段就是共享网络 conv1\_p 的日志了：

```
layer_factory.hpp:77] Creating layer conv1_p
net.cpp:92] Creating Layer conv1_p
net.cpp:428] conv1_p <- data_p
net.cpp:402] conv1_p -> conv1_p
net.cpp:144] Setting up conv1_p
net.cpp:151] Top shape: 64 20 24 24 (737280)
net.cpp:159] Memory required for data: 8721664
net.cpp:488] Sharing parameters 'conv1_w' owned by layer 'conv1', param
index 0
net.cpp:488] Sharing parameters 'conv1_b' owned by layer 'conv1', param
index 1
```

这一段最有特点的是最后两行以“Sharing”开头的日志。因为 Siamese Model 中拥有参数完全相同的两个网络，所以在构建第二个网络时 Caffe 检测到参数名字已经存在，说明该层的参数和其他层共享，于是它在这里输出日志告诉用户这一点。实际上，Net 类中还负责了参数相关的一系列初始化，除了参数共享的设定，还有对参数的 learning\_rate、weight\_decay 的设定。

接下来是模型中最特别的一层：Loss 层。

```
[LINE1] net.cpp:92] Creating Layer loss
[LINE2] net.cpp:428] loss <- feat
[LINE3] net.cpp:428] loss <- feat_p
[LINE4] net.cpp:428] loss <- sim
[LINE5] net.cpp:402] loss -> loss
[LINE6] net.cpp:144] Setting up loss
[LINE7] net.cpp:151] Top shape: (1)
[LINE8] net.cpp:154]         with loss weight 1
[LINE9] net.cpp:159] Memory required for data: 10742020
```

这一层看上去没有什么特别，该有的和前面一样，唯一不同的就是它的倒数第二行，这一层的 loss\_weight 值为 1，说明它是一个 Loss Layer。关于 loss\_weight 的问题，5.4 节已经介绍过，这里就不再赘述了。

前面的网络构建日志主要展示了网络组装的内容，从日志中可以看出，Net 类完成了以下事情。

1. 实例化 Layer。
2. 创建 bottom blob、top blob。
3. 初始化 Layer，确定 top blob 维度。
4. 确定 Layer 参数的 learning\_rate、loss\_weight 等信息。
5. 确定 Layer 的参数是否共享，不共享则创建新的。

从上面的过程也可以看出，Net 保存了整个网络中所有的流动性变量（bottom blob，top blob），还有各层的参数的指针信息，参数的共享关系等信息。这样的好处是把网络的数据集中到了一起，对数据进行统一操作时更方便。

顺着日志往下看，Net 就要计算日志开始显示一个 Layer 是否需要反向传播的计算，这里可以截取一小段日志：

```
net.cpp:220] pool1 needs backward computation.
net.cpp:220] conv1 needs backward computation.
net.cpp:222] slice_pair does not need backward computation.
net.cpp:222] pair_data does not need backward computation.
net.cpp:264] This network produces output loss
net.cpp:277] Network initialization done.
```

一般来说，模型层都是需要做反向计算的，但是也会有两类层不需要做反向计算。

- 一类是数据层，就像上面日志第 4 行展现的那样。
- 另一类是人为固定的模型层，这个需求一般在 finetune 网络模型时出现。因为反向计算一般比前向计算慢，如果能够跳过一些 Layer 的反向计算过程，就可以节省时间，提高效率。

到此为止，网络组装的工作已经完成，后面的日志主要展示了网络训练过程中的一些信息，就暂时不去关注了。了解了这些，相信读者对 Net 组装有了大概的了解，再去查看它的代码时就会轻松些。Net 类中所有的成员变量与它们之间的关系，可以通过图 5-9 来理解。

再看看 Net 类里面的成员变量就能看出两者之间的对应关系。以下便是 Net 类中定义的成员变量，为了和上面的图对应，源代码的顺序做了一定的调整：

```
// 属于Net基本属性的四个变量
string name_;
Phase phase_;
size_t memory_used_;
bool debug_info_;
```

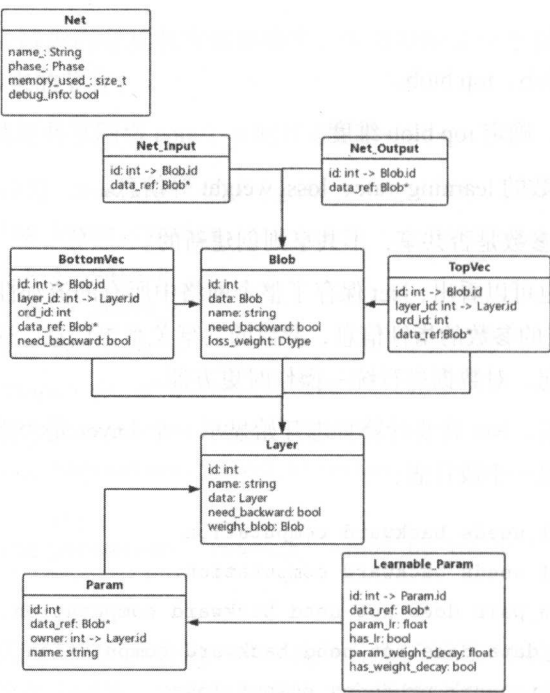


图 5-9 Caffe Net 的实体关系图

```
// Layer部分的变量
vector<shared_ptr<Layer<Dtype>>> layers_;
vector<string> layer_names_;
map<string, int> layer_names_index_;
vector<bool> layer_need_backward_;
/*
以下是网络中流动数据（top,bottom）的相关变量
*/
// 为了方便管理和共享，所有的top、bottom数据对象都保存在这里
vector<shared_ptr<Blob<Dtype>>> blobs_;
vector<string> blob_names_;
map<string, int> blob_names_index_;
vector<bool> blob_need_backward_;
vector<Dtype> blob_loss_weights_;
// bottom数据，这里保存的都是指针，原始数据在blobs_中保存
vector<vector<Blob<Dtype>*>> bottom_vecs_;
vector<vector<int>> bottom_id_vecs_;
```

```

vector<vector<bool> > bottom_need_backward_;
// top数据，这里保存的都是指针，原始数据在blobs_中保存
vector<vector<Blob<Dtype>*> > top_vecs_;
vector<vector<int> > top_id_vecs_;
// 网络的IO层的指针信息记录在这里
vector<int> net_input_blob_indices_;
vector<int> net_output_blob_indices_;
vector<Blob<Dtype>*> net_input_blobs_;
vector<Blob<Dtype>*> net_output_blobs_;
/*
以下是参数相关的变量
*/
// 参数相关的信息，这里的参数保存的也是指针，原始数据在各个Layer中保存
vector<shared_ptr<Blob<Dtype> > > params_;
vector<vector<int> > param_id_vecs_;
vector<int> param_owners_;
vector<string> param_display_names_;
vector<pair<int, int> > param_layer_indices_;
map<string, int> param_names_index_;
// 可学习的参数信息
vector<Blob<Dtype>*> learnable_params_;
vector<int> learnable_param_ids_;
// 每个参数自己的学习率
vector<float> params_lr_;
vector<bool> has_params_lr_;
// 每个参数自己的正则项权重
vector<float> params_weight_decay_;
vector<bool> has_params_decay_;

```

到此，Net 组装的内容就介绍完了。Net 类几乎包含了所有的运算实体，了解了它包含的内容，读者就会对模型计算中涉及的数据有较深刻的认识。

## 5.6 Caffe 的 Solver 计算流程

Caffe 中的模型训练简单来说只有两个步骤——前向后向计算和参数更新，了解了前向后向计算后，下面就介绍参数更新的过程。由于实际工程需要考虑的细节比较多，



这部分变得有些烦琐，本节首先介绍 Caffe 的优化流程，再介绍单机多卡的并行优化流程。

### 5.6.1 优化流程

真正的训练在 Solver 的 Step 函数内，去掉一些相对不重要的内容，它的核心代码如下所示：

```
void Solver<Dtype>::Step(int iters) {
    while (iter_ < stop_iter) {
        net_->ClearParamDiffs();
        for (int i = 0; i < callbacks_.size(); ++i) {
            callbacks_[i]->on_start();
        }
        Dtype loss = 0;
        for (int i = 0; i < param_.iter_size(); ++i) {
            loss += net_->ForwardBackward();
        }
        loss /= param_.iter_size();
        UpdateSmoothedLoss(loss, start_iter, average_loss);
        for (int i = 0; i < callbacks_.size(); ++i) {
            callbacks_[i]->on_gradients_ready();
        }
        ApplyUpdate();
        ++iter_;
    }
}
```

这段代码有五个核心函数，其中 on\_start() 和 on\_gradient\_ready() 是和多卡训练相关的回调函数，具体内容将在 5.6.2 节介绍，另外还有三个重要的过程：ForwardBackward、UpdateSmoothedLoss 和 ApplyUpdate。

ForwardBackward 函数调用了 Net 中的代码，主要完成了模型的前向后向计算，前向用于计算模型的最终输出和 Loss，后向用于计算每一层网络和参数的梯度。

UpdateSmoothedLoss 函数主要用于平滑模型产生的 Loss 值。Caffe 中参数的训练方式通常是基于批量数据的优化方法，由于批量数据的数量不够多，尤其在优化早期，每轮迭代产生的 Loss 差距可能很大，为了不让用户看到异常的 Loss 值而感到意外，Caffe

对 Loss 值进行了平滑操作，这样如果有个别迭代轮数的 Loss 比较大，Caffe 就会将其平滑处理，这样用户看到的平滑后的 Loss 更能代表模型的整体表现。

最复杂的就是 ApplyUpdate 函数，这个函数真正完成了参数更新的任务。下面就详细介绍这个函数的详细内容，它的核心代码如下所示：

```
void SGDSolver<Dtype>::ApplyUpdate() {
    Dtype rate = GetLearningRate();
    ClipGradients();
    for (int param_id = 0; param_id < this->net_->learnable_params().size();
        ++param_id) {
        Normalize(param_id);
        Regularize(param_id);
        ComputeUpdateValue(param_id, rate);
    }
    this->net_->Update();
}
```

第一个函数是 GetLearningRate，关于 learning rate 更多的故事将在 8.1 节详细介绍，在 CNN 训练过程中，选择合适的 learning rate 是个大问题。一般来说，随着迭代轮数不断增多，learning rate 应该逐渐变小。为了让 learning rate 的配置更灵活，Caffe 提供了一系列选择 learning rate 的方案。

- **fixed**: lr 永远不变
- **step**:  $lr = \text{baselr} \cdot \text{gamma}^{\text{iter}/\text{stepsize}}$
- **exp**:  $lr = \text{baselr} \cdot \text{gamma}^{\text{iter}}$
- **inv**:  $lr = \text{baselr}(1 + \text{gamma} \cdot \text{iter})^{-\text{power}}$
- **multistep**: 直接写 iter 在某个范围内时 lr 应该是多少
- **poly**:  $lr = \text{baselr}(1 - \frac{\text{iter}}{\text{maxiter}})^{\text{power}}$
- **sigmoid**:  $lr = \text{baselr} \frac{1}{1 + e^{-\text{gamma}(\text{iter} - \text{stepsize})}}$

读者可以从上面的方案中选择一个。

第二个函数是 ClipGradients，这一步对梯度值的大小做限制。如果梯度值过大，函数就会对梯度做一个修剪，对所有的参数乘以一个缩放因子，使所有参数的平方和不超过某个设定的上限。这个功能像是对优化函数设置了一个 Trust Region，可以防止参数更新量过大而导致无法预料的事情发生。这个函数主要用于防御性的修正梯度，并不能用于自适应地修正梯度大小。因为在实际训练过程中，参数的数值大小很可能并

不平均，有些参数的梯度比较大，有些参数的梯度比较小，那么对所有的参数乘以相同的因子会让一些本来比较小的参数变得更小，使得这些参数的训练变得更缓慢。所以这个方案只是一个保守解决问题的方案，想要自适应地解决问题，最好采用其他方案。

接下来就进入了每一个参数自己的训练过程。首先是 Normalize 方法，这一步要完成 Batch 级梯度数据平滑，它要让每一个参数的梯度除以 `iter_size` 参数。这主要考虑了单一 Batch 数据量不足的场景，代码比较简单。

接下来是 Regularize 方法，到这一步要完成正则项梯度的计算。Caffe 提供了两种正则方法——L2 和 L1，其中 L2 正则则是可导的，所以就采用了标准的梯度下降方法进行优化；而 L1 在原点处不可导，Caffe 采用了 sub-gradient 的计算方法。这个函数将对应正则项的梯度加到参数的梯度数据中。L2 正则的优化计算比较简单，直接求导进行计算：

```
# L2 Norm gradient update
diff += local_decay * data
```

但是 L1 正则的计算还是有值得玩味的地方的：

```
# L1 Norm gradient update
diff += local_decay * sign(data)
```

关于这个问题我们将在 8.4 节介绍。

最后是 ComputeUpdateValue 方法。这时梯度的计算已经完成，下面该考虑如何把 learning rate 和梯度结合起来计算最终的梯度优化值了。最经典的 SGDSolver 类采用基于动量的优化方法。除此之外，Caffe 还提供了一系列的梯度计算方法，这些优化方法各有特点，这些内容将在 8.3 节详细介绍。

到此为止，每一个参数的更新量都已经计算好，最后一步就是要调用 Blob 中的 Update 把每个参数的 data 和 diff 相加，将参数更新。这样，整个优化过程就完成了。

### 5.6.2 多卡优化算法

对于一些小任务来说，Caffe 的单卡训练已经可以满足需求，比方说一直提到的 MNIST 数据集。不过对于那些大规模的数据集来说，多卡训练是必不可少的，因为多卡训练可以减少模型的训练时间，所以本节就接着前面介绍 Caffe 的多卡训练。

Caffe 的多卡训练算法的总体思路是数据并行，在训练时数据被划分成几个部分，每一个部分用各自关联的 GPU 进行前向后向计算，然后将所有 GPU 的计算得到梯度汇总，最终完成更新。为了完成上面设想的步骤，Caffe 使用 Solver 的 Step 函数中提供

的两个回调函数，多卡训练也主要通过了这两个回调函数完成数据传输的工作，它的整体流程如下。

- 1. 回调函数 on\_start(): 将（更新后的）参数复制到每一个 GPU 中，使每个 GPU 拥有最新的参数。
- 2. ForwardBackward(): 每个 GPU 各自完成不同数据前向后向的计算，得到每个参数的梯度。
- 3. on\_gradient\_ready(): 将每个 GPU 中参数的反向梯度汇总到一个 GPU 所在线程上。
- 4. ApplyUpdate(): 在汇总的线程上进行参数的整体更新。

其中第 2 步由每一个 CPU 线程和与自己绑定的 GPU 并行完成，第 4 步由汇总的 CPU 和与自己绑定的 GPU 完成，第 1 步和第 3 步主要是完成数据传输的任务，也是多卡计算与单卡计算不同的地方。

现代的计算机架构中，多个 GPU 通常被连接到同一块主板上，就像多核 CPU 一样，这样计算机就可以通过并发访问多 GPU 获得更强大的计算能力。由于当前硬件架构的限制，GPU 之间不能直接相连，而是通过其他方式链接在一起。它们的结构如图 5-10 所示。

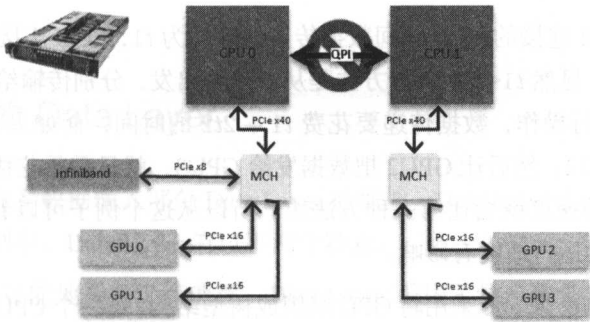


图 5-10 CPU-GPU 架构图（图片来源：<http://exxactcorp.com/blog/exploring-the-complexities-of-pcie-connectivity-and-peer-to-peer-communication/>）

有些 GPU 距离比较近，有些 GPU 距离比较远，因此不同 GPU 之间的数据传递速度是不一样的。属于同一组的 GPU 之间传输速度非常快，而不与同一 CPU 相连的 GPU 之间传输数据就会慢不少。在上面提到的第 1 步和第 3 步中，多个 GPU 之间的数据需要进行传递，那就要考虑 GPU 之间的通信问题。

nvidia 的命令：nvidia-smi 可以帮助查看使用的机器上 GPU 的组织架构。假设在某台机器上调用该命令后的显示结果如下所示：

```
nvidia-smi topo -m
```

# 以下为结果显示

	GPU0	GPU1	GPU2	GPU3
GPU0	X	PHB	SOC	SOC
GPU1	PHB	X	SOC	SOC
GPU2	SOC	SOC	X	PHB
GPU3	SOC	SOC	PHB	X

从中可以看出，这台机器有 4 块 GPU，它们之间存在着某种拓扑结构，他们的结构具体是什么样的呢？调用的命令包含了对表格中内容的解释：

Legend:

- X = Self
- SOC = Path traverses a socket-level link (e.g. QPI)
- PHB = Path traverses a PCIe host bridge

这个解释对于不了解硬件知识的读者实际上毫无意义，这里可以简单地理解为 PHB 表示数据间的传输速度非常快，SOC 表示数据传输需要穿越不同的 CPU（QPI 是连接不同 CPU 的桥梁），速度很慢。那么问题来了，如果想把一份数据从 GPU0 传递到其他三个 GPU 中，该如何传递？

这里假设 PHB 连接的 GPU 之间数据传递的时间为  $t_1$ ，SOC 连接的 GPU 之间数据传递的时间为  $t_2$ ，显然  $t_1 < t_2$ 。一种方法是从 GPU0 出发，分别传输给 GPU1、GPU2 和 GPU3。不考虑并行操作，数据传递要花费  $t_1 + 2t_2$  的时间，而如果只把数据从 GPU0 发给 GPU1 和 GPU2，然后让 GPU2 把数据发给 GPU3，就只需要花费  $2t_1 + t_2$  的时间，这样第二种方案的速度就会比第一种方法快。所以从这个例子可以看出，不同的数据传输方法对算法的运行速度有影响。

基于上面的结论，Caffe 采用将 GPU 组织成树型结构，每一个 CPU 线程和一个 GPU 绑定的方案。其中一个线程和这个线程对应的 GPU 作为树形结构的根，其他的 GPU 作为根下面的节点。所有的数据只能通过相邻的节点传递，如果两个 GPU 不相邻，那么数据传递就要通过其他节点进行。为了更快地传输 GPU 数据，树形结构的构建要考虑 GPU 之间是否相近，两个 GPU 之间是否可以进行点对点的数据直传这样的问题，回答这个问题只需要调用对应的 API 即可，CUDA 中有这样的方法：

```
int cudaDeviceCanAccessPeer(int, cudaDeviceProp, cudaDeviceProp)
```

这个函数可以通过检查两个 GPU 的属性告诉读者它们是否可以直连传输数据。

了解了上面的内容，优化步骤中的第 1 步和第 3 步就变得简单了。图 5-11 所示为上面例子中 GPU 架构下 Caffe 构建的树形结构及数据传输的流动图。

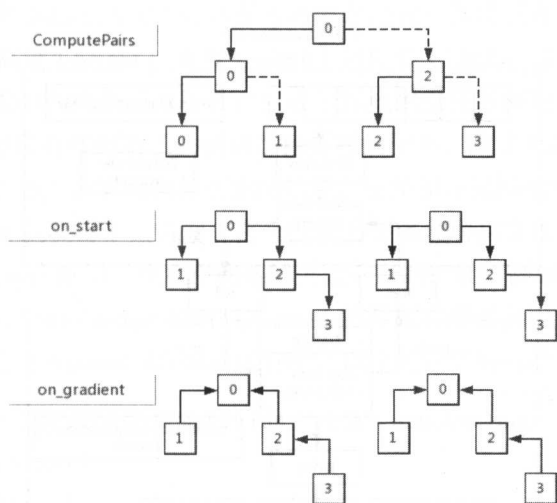


图 5-11 P2PSync 的关键步骤图示

其中 on\_start 和 on\_gradient 中的左右两幅图表示了数据传输的两个步骤。这样数据传递的问题就解决了，感兴趣的读者可以根据上面介绍的过程阅读代码，这里就不再详细讲述了。

## 5.7 Caffe 的 Data Layer

下面我们来看看数据输入部分 Data Layer 的框架。Data Layer 的主要功能是将数据从数据库读入模型中。Data Layer 有以下两个特点。

1. 为了能够尽可能地提高训练速度，Data Layer 采用了异步准备数据的形式，数据读入的工作和模型训练的工作在各自的线程中进行，相互独立并不依赖。当模型需要数据时，只需要将数据复制到指定的内存中即可。
2. 存入数据库的数据代表了某种数据实体，随着数据内容的不同，数据实体也可以随之改变以适应不同的需求。Data Layer 模块的整体关系图如图 5-12 所示。

下面就详细介绍里面的机制。

### 5.7.1 Datum 结构

首先来看看数据的表示实体 Datum。Datum 是输入数据的存储类，数据库中的数据以这个结构进行保存。以下就是这个类的 Protobuf 代码：

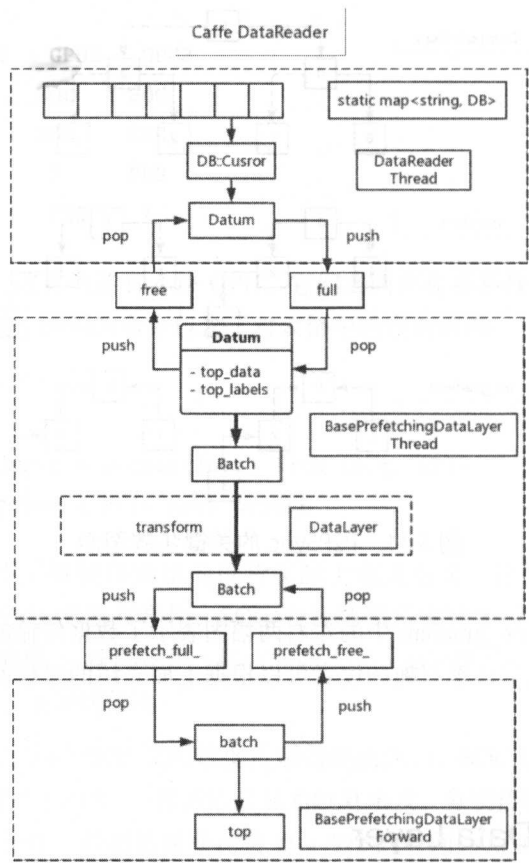


图 5-12 Data Layer 模块架构

```
message Datum {
  optional int32 channels = 1;
  optional int32 height = 2;
  optional int32 width = 3;
  // the actual image data, in bytes
  optional bytes data = 4;
  optional int32 label = 5;
  // Optionally, the datum could also hold float data.
  repeated float float_data = 6;
  // If true data contains an encoded image that need to be decoded
  optional bool encoded = 7 [default = false];
}
```

可以看出, 这种 Datum 结构主要的服务对象是经典的图像任务, 它的属性中包含了通道 (channel)、高度 (height)、宽度 (width) 和标签 (label), 这些属性可以满足我们完成经典图像分类的训练和预测任务; 但对于其他图像任务来说, 使用这个数据结构存储训练数据就显得有些困难了。例如目标检测的任务, 除了上面提到的这些属性, 我们还需要存储多个 ROI 位置和物体类别的信息, 这使得存储结构变得比现在复杂些, 那么直接使用 Datum 就显得不太方便。这个问题有很多种解决方法, 其中一种是继续使用 Caffe 的 Data Layer 框架, 并且在此架构上做一定的扩展。例如, 一个知名的图像物体检测的网络结构 SSD (Single Shot Detector) 的开源实现便采用了扩展 Datum 的方式, 它的训练数据是用 AnnotatedDatum 保存的, 以下是它扩展的数据结构:

```
// An extension of Datum which contains "rich" annotations.
message AnnotatedDatum {
  enum AnnotationType {
    BBOX = 0;
  }
  optional Datum datum = 1;
  // If there are "rich" annotations, specify the type of annotation.
  // Currently it only supports bounding box.
  // If there are no "rich" annotations, use label in datum instead.
  optional AnnotationType type = 2;
  // Each group contains annotation for a particular class.
  repeated AnnotationGroup annotation_group = 3;
}
```

### 5.7.2 DataReader Thread

DataReader 是 Caffe 用于从两种 DB 中读取数据的类, 它的目标仅仅是把数据从 DB 中读取出来, 也就是图 5-12 最上面的框内所展示的内容。DataReader 同时考虑到了系统的扩展性和并发性, 如果模型配置中设置了多个数据源, 那么 Caffe 会为每个数据源创建一个线程分别读取数据, 以实现数据的高效独立读取。如果模型采用多 GPU 的模式训练, 那么训练过程中程序中 will 拥有多个 Solver 类, 读取数据的线程将同时为这些 Solver 服务, 并保持线程间数据的独立。

最终在每一个 Solver 里的 Data Layer 都会有一个自己的 DataReader 对象, 它采用生产者—消费者的形式读取线程中准备好的数据。这其中会有一对关键的线程安全队列 BlockingQueue——free\_ 和 full\_:



```
class QueuePair {
public:
    BlockingQueue<Datum*> free_;
    BlockingQueue<Datum*> full_;
}
```

其中 free\_ 队列中保存着空闲的数据对象 Datum，full\_ 中保存了已经装有数据的对象。在 DataReader 的线程中，每一轮迭代将为所有 Solver 从数据库读取数据并存放在队列 full\_ 中：

```
void DataReader::Body::read_one(db::Cursor* cursor, QueuePair* qp) {
    Datum* datum = qp->free_.pop();
    datum->ParseFromString(cursor->value());
    qp->full_.push(datum);
    cursor->Next();
}
```

读取到的数据将被下游的 BasePrefetchingDataLayer 的线程（后面会提到）取走。这样只要 DataReader 能够保证从数据库中取出的数据供应充足，由于线程间异步操作的原因，读取数据所造成的时间消耗就将被节省下来。

### 5.7.3 BasePrefetchingDataLayer Thread

上面的 DataReader 已经把数据读出来了，而 BasePrefetchingDataLayer 类要做的就是数据的加工。这一部分主要完成两件事。

1. 确定数据层最终的输出形式（可以不输出 label 的）。
2. 完成数据层预处理（调用 Data Transformer 完成一些白化数据的简单工作，例如减均值、乘系数）。

BasePrefetchingDataLayer 是 Caffe 中的一个抽象层，它拥有前面提到的 Layer 的一切属性与功能。它的内部有一个线程，也拥有一对阻塞队列模拟生产者—消费者模型：

```
BlockingQueue<Batch<Dtype>*> prefetch_free_;
BlockingQueue<Batch<Dtype>*> prefetch_full_;
```

其中 prefetch\_free\_ 保存已经消费完的数据，prefetch\_full\_ 保存准备就绪等待消费的数据。这个线程负责把数据从 DataReader 中读出并进行数据处理，然后将处理好的数据放入准备队列中：

```

while (!must_stop()) {
    Batch<Dtype>* batch = prefetch_free_.pop();//取出一个空的对象
    load_batch(batch);//从DataReader读取数据并进行处理
    prefetch_full_.push(batch);//将准备好的数据放入队列
}

```

这样在每次调用 Forward 方法时, BasePrefetchingDataLayer 将从自己的准备队列中取出数据, 将数据复制给 top\_data:

```

void BasePrefetchingDataLayer<Dtype>::Forward_cpu(
    const vector<Blob<Dtype>*>& bottom, const vector<Blob<Dtype>*>& top) {
    Batch<Dtype>* batch = prefetch_full_.pop();
    top[0]->ReshapeLike(batch->data_);
    caffe_copy(batch->data_.count(), batch->data_.cpu_data(), top[0]->
    mutable_cpu_data());
    top[1]->ReshapeLike(batch->label_);
    caffe_copy(batch->label_.count(), batch->label_.cpu_data(), top[1]->
    mutable_cpu_data());
    prefetch_free_.push(batch);
}

```

#### 5.7.4 Data Layer

看上去 BasePrefetchingDataLayer 完成了所有的任务, 那为什么还说它是一个抽象层呢? 实际上在上面的线程代码中, 第 2 句调用的 load\_batch 在 BasePrefetchingDataLayer 中是一个虚函数。它把这个函数留给子类去实现——又是一个模板设计模式的例子。实现这个虚函数的是我们的正主——Data Layer, 以下是 Data Layer 实现的 load\_batch 方法:

```

void DataLayer<Dtype>::load_batch(Batch<Dtype>* batch) {
    Datum& datum = *(reader_.full().peek());
    Dtype* top_data = batch->data_.mutable_cpu_data();
    Dtype* top_label = batch->label_.mutable_cpu_data();
    for (int item_id = 0; item_id < batch_size; ++item_id) {
        Datum& datum = *(reader_.full().pop());
        int offset = batch->data_.offset(item_id);
        this->transformed_data_.set_cpu_data(top_data + offset);
    }
}

```

```

    this->data_transformer_->Transform(datum, &(this->transformed_data_));
    top_label[item_id] = datum.label();
    reader_.free().push(const_cast<Datum*>(&datum));
}
}

```

可以看出，在 Data Layer 中，数据从 DataReader 读出，并完成了数据的预处理工作。如果未来我们对数据进行更多、更复杂的操作，就可以在子类中实现自己的 load\_batch 方法控制流程。

Data Layer 的整体结构到这里就介绍得差不多了。通过 3 个类的交互，Caffe 实现了异步读取数据和处理数据的过程，同时如果需要定制不同的数据结构，只需要在架构中增加新的数据结构和操纵方法就可以复用整个框架了。

## 5.8 Caffe 的 Data Transformer

在 Data Layer 中提到了数据预处理的工作，下面来看一看 Data Transformer 的实现内容。Caffe 支持两种编程语言——C++ 和 Python，由于面向的应用不同，Data Transformer 在两种编程语言下的实现有所不同，为此本书将分开介绍。一般来说，模型用 C++ 的代码做训练，用 Python 的代码做预测（当然用 Python 做训练的例子也越来越多）。在 C++ 中 Data Transformer 的功能在 Data Layer 中被调用，可以说它成为后者的一个小部分，而在 Python 中它是一个独立的部分。

### 5.8.1 C++ 中的 Data Transformer

C++ 中的 Data Transformer 是与 Data Layer 集成在一体的，也就意味着在异步读入数据的同时可以完成数据的预处理。Transformer 对应的参数在 caffe.proto 有它的定义，内容如下所示：

```

message TransformationParameter {
    # Scale: 给每个像素值乘以一个系数
    optional float scale = 1 [default = 1];
    # Mirror: 做一个x轴的翻转，多用于自然场景的图像上
    optional bool mirror = 2 [default = false];
    # Crop: 这是在训练过程中经常用到的一种增强数据的方式。在train的过程中
    Caffe会进行随机crop，在test的过程中只会保留中间的部分。

```

```

optional uint32 crop_size = 3 [default = 0];
# Mean: 给每个像素减去一个均值, 一般是让数据具有Zero-Center的性质。
optional string mean_file = 4;
repeated float mean_value = 5;

optional bool force_color = 6 [default = false];
optional bool force_gray = 7 [default = false];
}

```

虽然上面的定义是扁平的, 但是实际执行过程中这些处理是有一定顺序的, 以下是 Data Transformer 中的核心代码:

```

template<typename Dtype>
void DataTransformer<Dtype>::Transform(const Datum& datum,
                                       Dtype* transformed_data) {
    const string& data = datum.data();
    const int datum_channels = datum.channels();
    const int datum_height = datum.height();
    const int datum_width = datum.width();

    const int crop_size = param_.crop_size();
    const Dtype scale = param_.scale();
    const bool do_mirror = param_.mirror() && Rand(2);
    const bool has_mean_file = param_.has_mean_file();
    const bool has_uint8 = data.size() > 0;
    const bool has_mean_values = mean_values_.size() > 0;

    Dtype* mean = NULL;
    if (has_mean_file) {
        mean = data_mean_.mutable_cpu_data();
    }
    if (has_mean_values) {
        if (datum_channels > 1 && mean_values_.size() == 1) {
            for (int c = 1; c < datum_channels; ++c) {
                mean_values_.push_back(mean_values_[0]);
            }
        }
    }
}

```

```

int height = datum_height;
int width = datum_width;

int h_off = 0;
int w_off = 0;
// 计算crop后的offset
if (crop_size) {
    height = crop_size;
    width = crop_size;
    if (phase_ == TRAIN) {
        h_off = Rand(datum_height - crop_size + 1);
        w_off = Rand(datum_width - crop_size + 1);
    } else {
        h_off = (datum_height - crop_size) / 2;
        w_off = (datum_width - crop_size) / 2;
    }
}

Dtype datum_element;
int top_index, data_index;
for (int c = 0; c < datum_channels; ++c) {
    for (int h = 0; h < height; ++h) {
        for (int w = 0; w < width; ++w) {
            data_index = (c * datum_height + h_off + h) * datum_width + w_off
                + w;
            // 计算镜像的index
            if (do_mirror) {
                top_index = (c * height + h) * width + (width - 1 - w);
            } else {
                top_index = (c * height + h) * width + w;
            }
            if (has_uint8) {
                datum_element =
                    static_cast<Dtype>(static_cast<uint8_t>(data[data_index]));
            } else {
                datum_element = datum.float_data(data_index);
            }
        }
    }
}

```



同样地，这些功能也有一个约定的顺序，这里展示了它的核心逻辑：

```
def preprocess(self, in_, data):
    self.__check_input(in_)
    caffe_in = data.astype(np.float32, copy=False)
    transpose = self.transpose.get(in_)
    channel_swap = self.channel_swap.get(in_)
    raw_scale = self.raw_scale.get(in_)
    mean = self.mean.get(in_)
    input_scale = self.input_scale.get(in_)
    in_dims = self.inputs[in_][2:] # 图像的长宽信息
    if caffe_in.shape[:2] != in_dims:
        # 1. 先将图像缩放到指定长宽比例
        caffe_in = resize_image(caffe_in, in_dims)
    if transpose is not None:
        # 2. 对图像的维度做一个调整，一般会从H * W * C调整为C * H * W
        caffe_in = caffe_in.transpose(transpose)
    if channel_swap is not None:
        # 3. 调整通道顺序，有的模型是BGR顺序，有的是RGB顺序
        caffe_in = caffe_in[channel_swap, :, :]
    if raw_scale is not None:
        # 4. 第一次数据缩放
        caffe_in *= raw_scale
    if mean is not None:
        # 5. 减去均值
        caffe_in -= mean
    if input_scale is not None:
        # 6. 第二次数据缩放，两次数据缩放一般只使用一次，这里是为了适
        # 应不同的使用者
        caffe_in *= input_scale
    return caffe_in
```

Python 版本的 crop 和 mirror 功能在 `[CAFFE_HOME]/python/caffe/io.py` 中的 `oversample` 函数，不过它的逻辑也和 C++ 的逻辑不太一样：

```
def oversample(images, crop_dims):
    # 首先计算出原图尺寸和裁剪后的尺寸
    im_shape = np.array(images[0].shape)
    crop_dims = np.array(crop_dims)
```

```

im_center = im_shape[:2] / 2.0

# 计算裁剪可能的范围
h_indices = (0, im_shape[0] - crop_dims[0])
w_indices = (0, im_shape[1] - crop_dims[1])
crops_ix = np.empty((5, 4), dtype=int)
curr = 0
for i in h_indices:
    for j in w_indices:
        crops_ix[curr] = (i, j, i + crop_dims[0], j + crop_dims[1])
        curr += 1
crops_ix[4] = np.tile(im_center, (1, 2)) + np.concatenate([
    -crop_dims / 2.0,
    crop_dims / 2.0
])
crops_ix = np.tile(crops_ix, (2, 1))

# 这里限定了crop的数量是
crops = np.empty((10 * len(images), crop_dims[0], crop_dims[1],
    im_shape[-1]), dtype=np.float32)
ix = 0
for im in images:
    for crop in crops_ix:
        crops[ix] = im[crop[0]:crop[2], crop[1]:crop[3], :]
        ix += 1
    # 这里对一般的数据进行了mirror操作
    crops[ix-5:ix] = crops[ix-5:ix, :, ::-1, :] # flip for mirrors
return crops

```

为了展现它的功能，这里将以一个简单的小程序为例测试这个功能效果：随机找一张图像作为输入，然后用上面的函数生成一批结果并输出出来，代码如下所示：

```

import sys
import caffe
import skimage.io

img_path = sys.argv[1]
img = caffe.io.load_image(img_path, color=False)

```



```
imgs = [img]
crop_imgs = caffe.io.oversample(imgs, (20, 20))
for i, crop_img in enumerate(crop_imgs):
    res_img = skimage.io.imsave( '{}.jpg'.format(i), crop_img.squeeze())
```

图 5-13 所示为原图和生成出来的 10 张经过 crop 和 mirror 操作的图像。当然，这里只是拿这个数字图像做一个例子，实际中基本上没有人会对数字图像进行 mirror 操作。



图 5-13 Data Transformer 的结果

## 5.9 模型层扩展实践——Center Loss Layer

前面的章节中读者已经完整地了解了 Caffe 的内容，虽然已经足够详细，但毕竟只是站在一个旁观者的角度观察 Caffe 的样子，了解如何使用 Caffe。随着深度学习的发展，越来越多崭新的技术不断涌现，原来的 Caffe 框架已经不能满足新技术的全部需求，这就需要读者对 Caffe 进行扩展，使它跟上前进的步伐。本节将用一个具体的例子展示如何通过修改代码完成对 Caffe 框架的增强，从而实现一些更高级的功能。

本节选用的例子来自一篇十分有意思的文章 *A Discriminative Feature Learning Approach for Deep Face Recognition*<sup>[1]</sup>，这篇文章介绍了作者精心设计的一个用于用户提高类别的区分度的损失函数——Center Loss，并通过一系列的实验证明它有效。虽然作者也给出了自己的实现代码，但本节是从头开始把这个代码实现一遍，看看扩展 Caffe 代码的一般流程。

### 5.9.1 Center Loss 的原理

在写代码之前，读者需要先简单了解背景知识——也就是 Center Loss 的算法和作用。

首先，让我们把目光集中在图像分类问题上。大家都知道图像分类问题的数据由一对信息组成——一个是图像，另一个是图像的标签，问题的目标就是建立一个模型，这个模型可以通过分析图像得到对应的标签。通常大家对深度学习模型各层的功能有一个共识，模型前面几层网络主要用于分析抽取原始图像中的一些特征，等特征分析

完成后，模型的后面几层就可以判别拥有这些特征的图像应该属于哪个类别。一直以来，模型的控制主要集中在最终的损失函数和正则项上，没有对中间结果做控制，而这篇文章的主要目标就是增加对中间结果的控制，从而解决一个实际问题。

于是乎作者设计了一个全新的网络，这个网络用于处理 MNIST 手写数字数据集的分类问题。网络结构如图 5-14 所示。

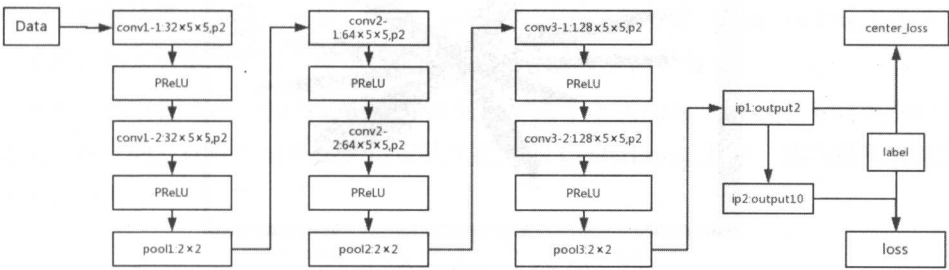


图 5-14 Center Loss 模型

这个网络和前面的 LeNet 相比有一定的不同。

首先是层数，LeNet 有 2 层卷积层和 2 层全连接层，而这个模型有 6 层卷积层和 2 层全连接层。

其次是倒数第二个全连接层。在这个模型中这一层的输出是一个 2 维的特征向量，和 LeNet 的向量维度有很大的不同。2 维特征向量有一个好处，那就是可以很容易地对其进行可视化，看看每一个数字的特征在 2 维空间下的分布情况。

实验分成如下两个步骤。

- 1. 对模型进行训练。
- 2. 用最终训练好的模型对 MNIST 的 Test 集合数据进行预测，并把倒数第二个全连接层的输出保存做可视化展示。

实验结果的可视化图像如图 5-15 所示。

图 5-15 和论文上给出的图像总体相似。读者一定会有如下两个明显的感觉。

- 1. 同一个类别内部的差距比较大。每个类别数字的分布有些分散，看上去都是长长的一条。
- 2. 类别之间的差异又比较小。几个类别相互交错在一起，很容易判断错误。

虽然最终的识别精度还是比较不错的，但是由于这两个问题的存在，这样的结果显然是让人不够满意的。对于第一个问题来说，模型并没有很好地找到同一类型图像的共同特征，从而导致同类型的图像的特征差异明显；对于第二个问题来说，有些同类型的图像特征之间的距离比不同类型的图像还大。对于某些基于距离比较的图像识别

来说，要在某个特征空间下通过找寻距离自己最近的向量判定自己所属类别，图 5-15 所示的特征抽取显然不够好，因此我们需要改进这个模型。

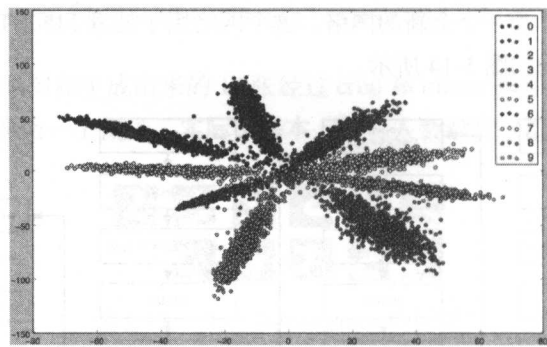


图 5-15 数据可视化

那么改进成什么样子就是好的模型呢？作者认为我们理想中的结果应该如图 5-16 所示。

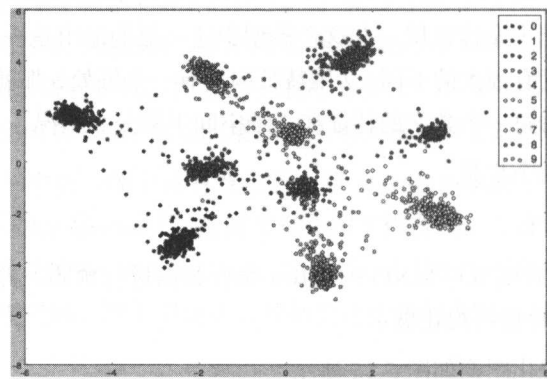


图 5-16 理想中的可视化效果

它很好地解决了我们前面遇到的两个问题：类内不够紧凑和类间距离太近。类内已经聚合得足够紧凑，而类间的距离也被拉得很开。这是怎么做到的呢？其中的主要功劳来自于作者设计的 Center Loss 给出的约束。

那 Center Loss 是什么模样呢？这个模型要给每一类数据定义一个中心点，这个中心点和聚类问题中的中心点十分相似。模型的目标是希望同一类数据计算出来的特征都能靠近自己类别的中心点，而不要远离这个中心点；离得近的惩罚小，离得远的惩罚

大, 于是最基本的 Center Loss 公式就出现了:

$$\text{CenterLoss} = \frac{1}{2N} \sum_{i=1}^N \sum_{k=1}^K I(\mathbf{x}_i \in \text{Class}_k) |\mathbf{x}_i - \mathbf{c}_k|_2^2$$

上面的公式中  $N$  表示一个 batch 内数据的总量,  $K$  表示求解问题的类别总数,  $\mathbf{x}_i$  表示一个数据计算出的特征向量,  $\mathbf{c}_k$  表示第  $k$  类别的中心向量,  $I(\mathbf{x}_i \in \text{Class}_k)$  是一个指示函数, 如果数据属于这个类别, 那么指示函数为 1, 否则为 0。

这里也可以把上面的损失函数做一点简化, 由于上面的损失函数中不同类别的数据是相互独立的, 计算时只要关注指定某个类别的  $\text{CenterLoss}_k$  就好, 然后将所有类的损失汇总即可:

$$\text{CenterLoss}_k = \frac{1}{2M} \sum_{i=1}^M |\mathbf{x}_i - \mathbf{c}_k|_2^2$$

其中  $M$  表示当前 batch 中这个类别的数据数量, 其余变量与上面的公式一致, 这样看上去就简单了不少。

这个损失函数看上去还是很有道理的, 但是还有些细节问题需要解决。例如, 每个类别的中心向量如何求出呢? 最简单直接的方法就是对上面的公式求偏导, 求偏导的结果如下:

$$\frac{\partial \text{CenterLoss}_k}{\partial \mathbf{c}_k} = \frac{1}{M} \sum_{i=1}^M (\mathbf{c}_k - \mathbf{x}_i) = \mathbf{c}_k - \frac{1}{M} \sum_{i=1}^M \mathbf{x}_i$$

从结果来看, 偏导相当于把每一个 batch 特征向量的平均值和当前中心的向量求差。最终的效果相当于用所有向量的平均值表示中心向量, 这个结果同样很有道理。

看上去这个损失函数已经没有太大问题了, 但是接下来还要解决一些实际运算中的问题, 由于每个 batch 的数据量并不算多, 如果直接按照上面的公式更新, 很容易因为数据误差导致中心点产生抖动, 这样会不利于最终优化的效果。面对这样的问题, 作者的方法是给梯度再加个缩放因子  $\alpha$ , 类似于优化中的步长, 让这个更新量不要太大, 这里  $\alpha$  肯定是不大于 1 的 (敢于尝试的小伙伴可以试试大于 1 的效果), 于是公式就变成了:

$$\Delta \mathbf{c} = \frac{\alpha}{N} \sum_{i=1}^N (\mathbf{c} - \mathbf{x}_i)$$

现在 Center Loss 的问题解决了, 模型优化目标有两个函数了, 那两个 Loss 之间的权重该怎么平衡呢? 最简单的方法就是再加一个超参数  $\lambda$ , 用于控制两个损失函数之间的比例。到此为止我们关于 Center Loss 的介绍就结束了, 经过训练实验, 模型确实产生了图 5-16 所示的理想结果, 效果令人满意。

### 5.9.2 Center Loss 实现

了解了算法的原理，下面就来从实战的角度看看如何修改 Caffe 的源码，让它拥有这个新的功能。代码修改的过程分成下面几个部分。

1. 定义 Center Loss Layer。
2. 实现 Center Loss Layer。
3. 完成训练 Center Loss Layer 的配置文件。
4. 完成预测 Center Loss Layer 的配置文件。
5. 完成可视化的脚本。

下面将一一介绍。

定义 Center Loss Layer 的工作在 Protobuf 文件中完成，首先在 *src/caffe/proto/caffe.proto* 中加入配置定义的内容，这样在网络结构定义的配置文件中就可以使用 Center Loss Layer 了：

```
message LayerParameter {
    optional CenterLossParameter center_loss_param = 200;
}

message CenterLossParameter{
    optional float loss_weight = 1 [default = 0.003];
    optional float alpha = 2 [default = 0.5];
    required uint32 cluster_num = 3;
}
```

其中上面的第一段补充在 LayerParameter 中，第二段是 CenterLoss 的参数，其中 *loss\_weight* 就是平衡两个 Loss 的权重，一般还是会把 Center Loss 放在次要的位置，所以默认的权重值是 0.003，比较小；*alpha* 就是梯度的缩放因子；*cluster\_num* 告诉 Center Loss Layer 训练数据有多少个类别，这样方便它分配内存空间。

下面就是实现 Center Loss Layer 的代码，分别在 *include/caffe/layers* 和 *src/caffe/layers* 中放入模型层的头文件和定义文件。其中头文件的内容如下所示：

```
namespace caffe{
    /*
    NOTE: 此版本我们不存储中心向量
    */
    template<typename Dtype>
    class CenterLossLayer: public LossLayer<Dtype>{
```

```

public:
    explicit CenterLossLayer(const LayerParameter &param): LossLayer<Dtype>
    >(param){}
    virtual void LayerSetUp(const vector<Blob<Dtype>*> &bottom, const
    vector<Blob<Dtype>*> &top);
    virtual inline int ExactNumBottomBlobs() const {return 2;}
    virtual inline const char *type() const {return "CenterLoss";}
    virtual inline bool AllowForceBackward(const int bottom_index) const {
        return bottom_index != 1;
    }
    // 以下内容省略
}

```

上面的定义中有几个和框架关系比较紧密的地方。

第一个是 CenterLossLayer 要继承 LossLayer，这样它就会被 Caffe 自动识别成一个计算 Loss 的模型层。

第二个是告知框架这个层的一些约束：例如输入的数据只能有两个，这个层的名词，这个层的哪些内容将被反向传导。其中一个输入是特征数据，另一个是数据的标签，很显然标签不需要反向传导，这里就需要做一点限制。

实现代码里还有一些值得细说的事情：

```

namespace caffe{
    template<typename Dtype>
    void CenterLossLayer<Dtype>::LayerSetUp(const vector<Blob<Dtype>*>&
    bottom,
        const vector<Blob<Dtype>*>& top){
        LossLayer<Dtype>::LayerSetUp(bottom, top);
        CHECK_EQ(bottom[0]->num(), bottom[1]->num());
        // 以下省略具体的初始化过程
    }

    template<typename Dtype>
    void CenterLossLayer<Dtype>::Forward_cpu(
        const vector<Blob<Dtype>*> &bottom,
        const vector<Blob<Dtype>*> &top){
        const Dtype *feature = bottom[0]->cpu_data();
        const Dtype *label = bottom[1]->cpu_data();
    }
}

```

```

        int num = bottom[0]->num();
        int channels = bottom[0]->channels();
        // 以下是具体计算过程，结果保存在top.data()中
    }

template<typename Dtype>
void CenterLossLayer<Dtype>::Backward_cpu(
    const vector<Blob<Dtype>*> &top,
    const vector<bool> &propagate_down,
    const vector<Blob<Dtype>*> &bottom){
    int num = bottom[0]->num();
    int channels = bottom[0]->channels();
    // 以下是具体计算过程，结果保存在bottom.diff()中
}

template <typename Dtype>
void CenterLossLayer<Dtype>::Forward_gpu(
    const vector<Blob<Dtype>*>& bottom,
    const vector<Blob<Dtype>*>& top){
    Forward_cpu(bottom, top);
}

template<typename Dtype>
void CenterLossLayer<Dtype>::Backward_gpu(const vector<Blob<Dtype>*>&
top,
    const vector<bool>& propagate_down, const vector<Blob<Dtype>*>&
bottom){
    Backward_cpu(top, propagate_down, bottom);
}

INSTANTIATE_CLASS(CenterLossLayer);
REGISTER_LAYER_CLASS(CenterLoss);
}

```

由于它的算法比较简单，所以具体的计算抛开不管，主要关注代码架构的细节：Setup 方法完成初始化的工作；前向后向的计算主要是完成核心功能的计算，这里我们没有实现 GPU 的计算，只是让 GPU 的方法直接调用 CPU 的方法；最后的两句宏方法就是前面提到的模型层注册，这样 Caffe 就通过工厂方法创建了这个 Layer。

完成了定义后，再来看看模型的训练配置，其他配置和之前的代码相近，唯一不同的就是加入了 CenterLoss Layer：

```

layer {
  name: "center_loss"
  type: "CenterLoss"
  bottom: "ip1"
  bottom: "label"
  top: "center_loss"
  center_loss_param {
    loss_weight: 0.003
    alpha: 0.5
    cluster_num: 10
  }
}
}

```

预测的配置也基本一致，只是将 Loss 去掉就可以了。接着用 Python 接口进行预测，并将倒数第二层向量保存下来：

```

import caffe
import numpy as np
import lmdb
from caffe.proto import caffe_pb2
import matplotlib.pyplot as plt

if __name__ == '__main__':
    # 读取参数，这里省略了参数校验的步骤
    model_file = sys.argv[1]
    weight_file = sys.argv[2]
    lmdb_file = sys.argv[3]

    # 读取模型
    net = caffe.Net(model_file, weight_file, caffe.TEST)
    input_name = net.inputs[0]

    # 读取数据
    lmdb_env = lmdb.open(lmdb_file)
    lmdb_txn = lmdb_env.begin()
    lmdb_cursor = lmdb_txn.cursor()
    datum = caffe_pb2.Datum()

```



```

count = 0
batch_data = []
batch_label = []
for key, value in lmbd_cursor:
    datum.ParseFromString(value)
    label = datum.label
    data = caffe.io.datum_to_array(datum)
    im = data.astype(np.uint8)
    count += 1
    batch_data.append(im)
    batch_label.append(label)
    if count % 64 == 0:
        # 前向计算并输出向量信息
        data = np.array(batch_data, dtype=np.float32) / 255.0
        net.forward_all(**{input_name: data})
        res_data = net.blobs['ip1'].data
        for res, label in zip(res_data, batch_label):
            featStr = [str(f) for f in res]
            print str(label) + '\t' + '\t'.join(featStr)
        batch_data = []
        batch_label = []

```

之后利用保存的数据，读者就可以分析这些数据或者绘制上面的图像了。到这里一个完整的扩展过程就基本讲完了。

### 5.9.3 实验分析与总结

完成了这个实验，这里还要对实验的结果做一点简单的分析。这个模型在解决上面提到的两个问题上效果是很不错的，但是它最终的测试精度并不算高，只有 0.9888。比正常 LeNet 的 0.99 稍低一点。由于倒数第二个全连接层的特征维度被缩减成了 2，所以识别的精度肯定会受到影响，不过这只是为了可视化的效果，所以在真正的实验中可以把这个数字调大。

如果直接修改 LeNet 模型，将它的倒数第二层的维度减到 2，会造成模型无法收敛，所以论文中的 LeNet++ 使用了 6 层卷积，对于 MNIST 这样的小问题使用 6 层卷积加 2 层全连接也算是杀鸡用牛刀了。由于模型相对复杂，所以训练起来要费点时间。在实验中加入 Center Loss 后 Test Accuracy 实际上是下降了一点的。不过这点下降并不能说

明 Center Loss 对这个问题起了反作用，还是需要尝试倒数第二个全连接层的维度大于 2 时的情况。

上面提到的实现代码还存在一些瑕疵，那就是 Center Loss 的信息并没有被保存起来，这样如果模型需要断点训练，上一次计算的 Center 信息就会丢失，训练过程也就可能出现问题。实际上，这里最好把它保存在模型参数中，这部分留给各位读者做扩展。

总的来说，这个例子介绍了模型层扩展的方法，但没有涉及更多的修改。虽然从开发的角度来说并不算难，但是读者经常会遇到这类模型修改的问题。实际上，现在还有很多比这更复杂的例子，其中会涉及更多的修改，这部分就交给读者自行研究了。有很多方法能增强 Caffe 的功能，读者可以自学别人的研究成果去实现这些功能，也可以在开源社区寻找优秀的开源模块加入到自己的 Caffe 中，组装属于自己的、顺手的工具。

## 5.10 总结

本章主要介绍了深度学习开源框架 Caffe 的使用和源码分析，让我们来回顾一下。

- Caffe 的使用：从数据预处理到测试集数据验证的全过程的操作方法。
- 常用网络层的参数配置：卷积层、全连接层、非线性层、Pooling 层、Loss 层，以及 Data 层。
- Caffe 的源码结构：Layer、Net、Solver、Data Layer，以及 Data Transformer。
- 实现一个全新的 Layer：配置项定义、源代码编写，以及测试运行。

## 5.11 参考文献

[1] Wen Y, Zhang K, Li Z, et al. A Discriminative Feature Learning Approach for Deep Face Recognition[J]. Computers & Operations Research, 2016, 47(9):11-26.

# 6

## 深层网络的数值问题

机器学习的问题说到底都是数字游戏：问题被抽象成一个个数字，求解被抽象成数字的运算，结果同样被抽象成数字的组合……数字成了这门学科的核心。由于深度学习中的网络深度不断加大，数值运算也变得越来越复杂，所以就有必要聊聊数字运算中的一些事儿。本章首先回顾非线性层的内容，通过实验分析参数初始化、ReLU 函数对模型的影响；接下来，详细介绍两种当今常见的参数初始化方法——Xavier 和 MSRA 方法，并分析参数初始化和网络训练的关系；然后介绍训练数据预处理的方法——ZCA；最后谈谈模型计算中经常遇到的数值溢出及相应的解决办法。

### 6.1 ReLU 和参数初始化

本节的目标是分析非线性函数、参数初始化与模型数值的关系。前面的章节曾经介绍过 3 个非线性函数——Sigmoid、Tanh 和 ReLU。ReLU 函数相比于 Sigmoid 函数的优势如下。

- Sigmoid 整体的梯度值偏小，在反向传导的过程中会使梯度的幅度不断变小甚至消失；ReLU 的梯度对正向的接受域比较友好，不会损失反向回传的梯度值。
- Sigmoid 存在较严重的梯度消失问题：如果非线性函数的输出靠近 0 或者 1，那么 Sigmoid 的梯度会接近 0，反向传播的梯度传导到这里与 Sigmoid 的梯度相乘后也会接近 0，于是梯度到这里就消失了。ReLU 对负数的输出同样存在这样的问题，但是在正数输出方面表现比较好。

基于这两个原因，ReLU 这样的非线性函数在深度网络前向后向计算的传递性方面效果更好，更适合 CNN 这样层数比较深的网络，ReLU 更能够保证梯度信息的高质量传递。

但第 4 章中也曾提到，实际上 ReLU 也存在着一定的问题，下面就通过三个实验展示 ReLU 过宽的接受域对参数数值带来的挑战。实验的数据集依旧为 MNIST，模型也和 4.4 节所用的模型相同。不同的是这一次将关注更多细节上的内容，实验将关注下面这些数值信息。

- 1. 卷积层和全连接层的输出数据随优化变化的情况。
- 2. 卷积层和全连接层的参数数据随优化变化的情况。
- 3. 卷积层和全连接层的参数梯度随优化变化的情况。
- 4. 所有参数的数值、梯度的 L1、L2 范数随优化变化的情况。

6.1.1 第一个 ReLU 数值实验

第一个实验用到的非线性函数为 ReLU，经过 10000 轮迭代，最终的测试集精度为 0.991，下面就是由训练生成的日志完成的变化图。

首先是输出数据，如图 6-1 所示。图中仅展示了需要进行参数训练的四个模型层：两个卷积层 conv1、conv2；两个全连接层 ip1、ip2 的信息。

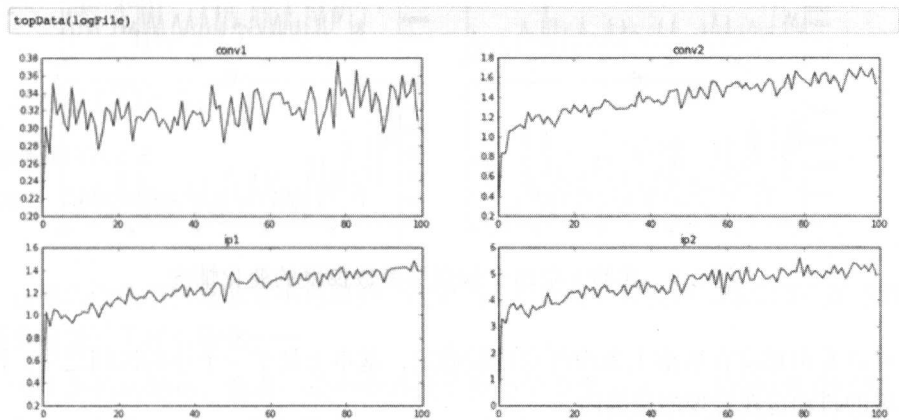


图 6-1 实验 1 中四个模型层输出数据的变化情况（横轴表示训练迭代轮数，纵轴表示对应变量的 L1 范数）

可以看出除了一开始的数值波动，后面的迭代中数值整体表现比较稳定，稳中有升，没有特别的震荡。

其次是参数的数值，如图 6-2 所示。

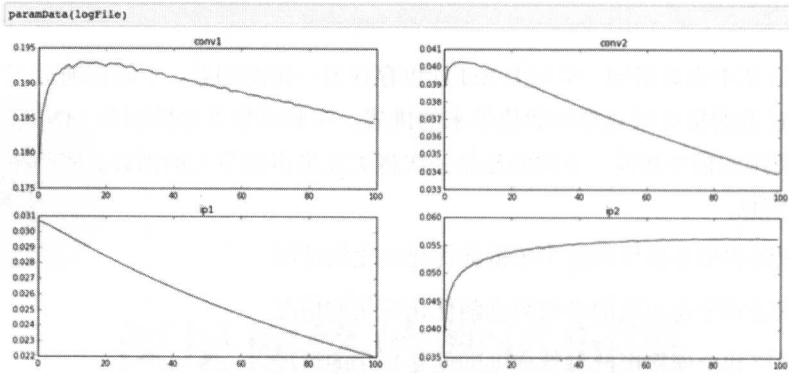


图 6-2 实验 1 中四个模型层参数数值的变化情况（坐标轴含义同图 6-1，以下同）

可以看出参数整体表现不算稳定，不过考虑到数据的绝对改变量不大，所以这个变动还算可以理解。

接下来是参数的梯度，如图 6-3 所示。

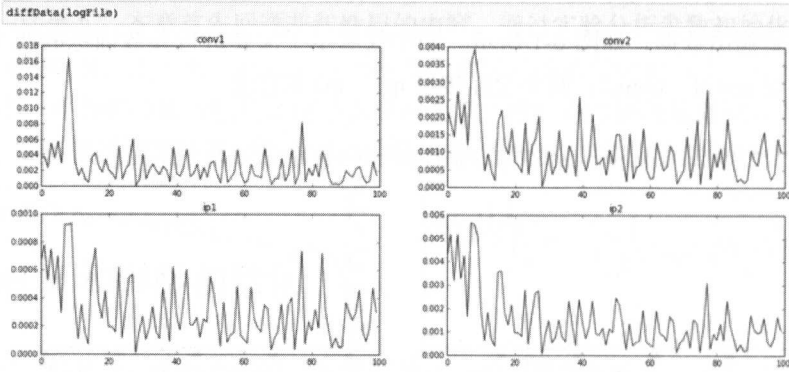


图 6-3 实验 1 中四个模型层参数梯度的变化情况

可以看出梯度在数值上表现得也比较稳定，基本上处于一个小的区间之中，图像中出现的抖动也在合理的范围内。

最后是全体参数的 L1 范数和 L2 范数，如图 6-4 所示。

这些数值和前面各层的表现基本一致。总体来说，各层输出和参数的梯度数值没有太大的波动，保持在一个稳定幅度上，所以整体的数值比较稳定，最终模型的表现也十分不错。

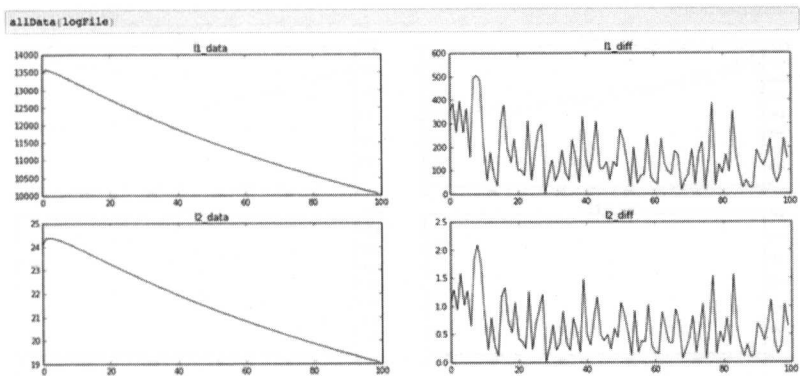


图 6-4 实验 1 中四个模型层参数的总体数值、梯度的 L1、L2 范数变化情况

6.1.2 第二个 ReLU 数值实验

看完了这个成功案例，下面就来看一个失败案例。当然，这个失败案例一般不会在现实中出现，只是做一个极端的效果而已，但这个效果也足以说明一些问题。这个实验不修改网络结构，只修改四个核心网络层（conv1、conv2、ip1、ip2）的参数的初始化方法。

原来的初始化方法是这样的：

```
weight_filler {
  type: "xavier"
}
```

现在将它们改成：

```
weight_filler {
  type: "gaussian"
}
```

这样的改法当然是非常不可取的，这相当于对所有参数使用均值为 0，方差为 1 的高斯分布进行采样初始化……

由于前面的铺垫，读者一定会觉得这个模型的效果会很差。10000 轮训练结束后，最终的精度仅为 0.1135。要知道 MNIST 中一共只有 10 个数字，随机猜也可以达到 0.1 的精度。这个模型采用了比较复杂的网络结构，却因为使用了不好用的参数初始化方法，导致结果只比猜稍微好一点，实在是太不应该了。这件事情非常值得认真思考。

那么，这个失败的模型在数值上的表现是怎样的呢？直接来看四张图，依次是各层的输出、参数数值、参数梯度、全体参数的 L1 和 L2 范数，如图 6-5~ 图 6-8 所示。

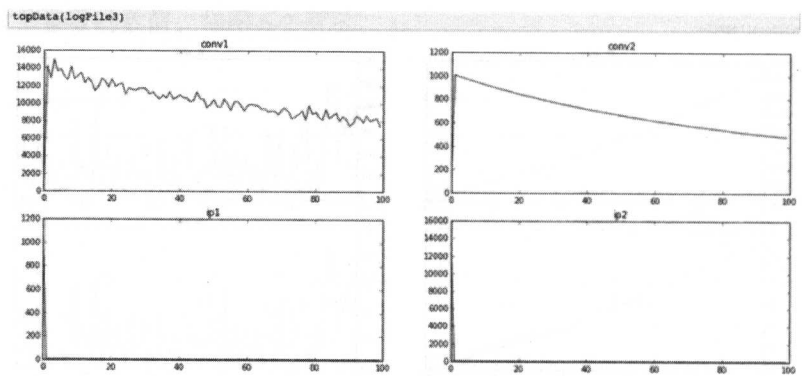


图 6-5 实验 2 中四个模型层输出的变化情况

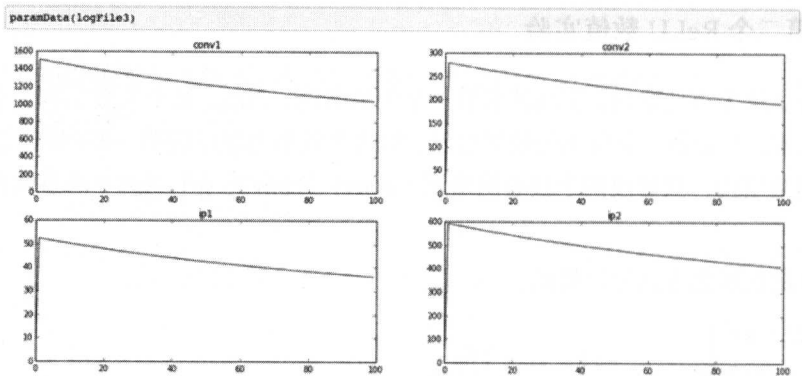


图 6-6 实验 2 中四个模型层参数数值的变化情况

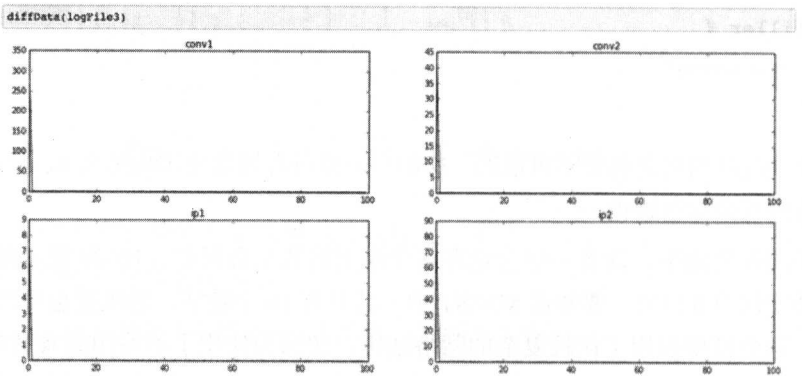


图 6-7 实验 2 中四个模型层参数梯度的变化情况

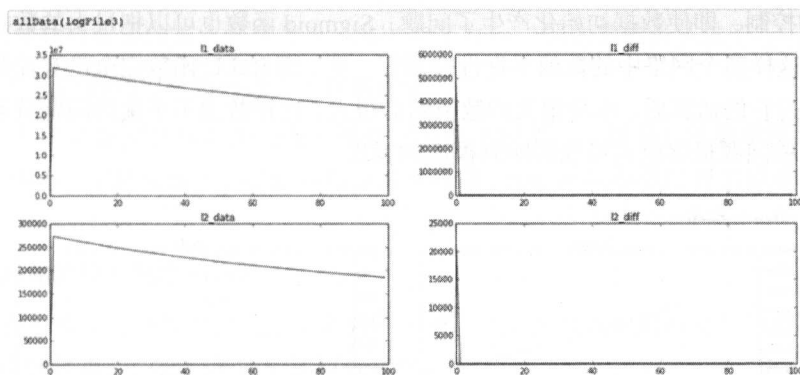


图 6-8 实验 2 中四个模型层参数的总体数值、梯度的 L1, L2 范数变化情况

可以看出,各个数据的数值在幅度上的差距都非常大,尤其是图 6-8,全体参数数值的 L1 范数已经达到  $2 \times 10^7$ ,而梯度却小到无法显示清楚,可见两者在数值上的差距非常大,所以在训练时,小量的梯度更新对大量的数值完全起不到明显的改变作用,因此模型从头至尾没有实质的改变,所以它的表现就和随机猜测差不多,因为模型并没有从训练数据中学到足够多的知识。

从上面的结果看来,模型训练失败一定是初始化不正确导致的。那么,所有的问题是不是全部由初始化造成呢?

### 6.1.3 第三个实验——Sigmoid

第三个实验就来看看初始化不正确是不是导致模型学习失效的根源。在保持上一个实验初始化设置的基础上,模型的 4 个核心层的非线性函数将全部换成 Sigmoid。Sigmoid 函数在第 4 章的实验中的表现落了下风,那么在这次的实验中它能否实现逆袭呢?

使用 Sigmoid 的模型在测试数据集上的最终结果为: 0.9538,虽然这个精确率不能算特别高,但是它起码说明模型还是从训练数据中学到了很多有用的知识。和 ReLU 的 0.11 相比,简直是天壤之别。这里只展示全体参数 L1、L2 范数的图,如图 6-9 所示。

虽然参数数值的幅度很大,但是梯度的幅度也不算小,对于参数的更新还是具有显著性的,至少比第 2 个实验中的 ReLU 模型强不少。这么看来,第 2 个实验的失败和 ReLU 有关,也和初始化方法有关。实际上,失败的原因是这两个方法没有搭配好。

实验前曾提到,ReLU 拥有宽广的接受域,这既是一件好事儿,也可能是一件坏事儿。读者都已经知道 Sigmoid 可以把任意的输入数据压缩到 0~1,所以在使用 Sigmoid 函数时,不用太担心数据的幅度问题,因为只要使用一个 Sigmoid,数据的幅度就会得



到良好的控制，即使数据初始化产生了问题，Sigmoid 函数也可以把过大的数据转换到小范围，这样整个网络中的数值平衡性会更好一些。而 ReLU 函数完全没有控制数据的幅度。经过它的运算后，本身很大的数依然会很大，这样数值不平衡的问题就不会得到缓解，这些问题最终就有可能影响到模型的表现。

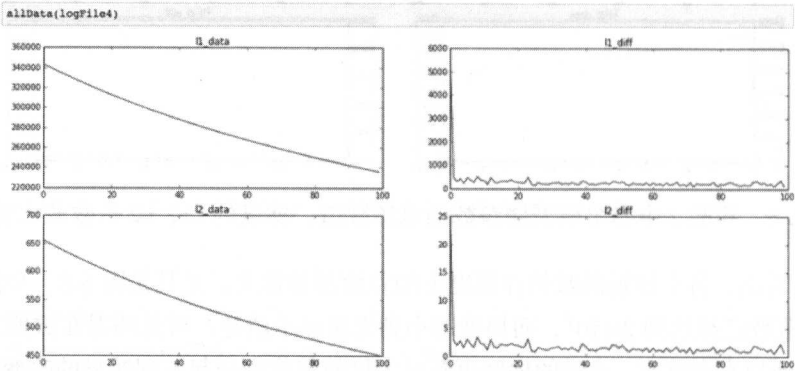


图 6-9 实验 3 中四个模型层参数的总体数值、梯度的 L1、L2 范数变化情况

经过这个实验，读者可以总结出如下经验。

1. Sigmoid 函数在控制数据幅度方面有优势，在深层网络中，使用 Sigmoid 可以保证数据幅度不会出现大的问题；但是 Sigmoid 函数存在梯度消失的问题，在反向传播上有劣势，所以在优化的过程中存在训练不足的情况。
2. ReLU 函数不会对数据的幅度做控制，那么在深层模型中，数据的幅度有可能产生一定的不平衡，最终会影响模型的表现；但是 ReLU 在反向传导方面可以很好地将“原汁原味”的梯度传递给后面的参数，这样优化的效果也会更好。

要评判哪个非线性函数更好，不但要看自己本身，还要看它们和模型配置的搭配情况。本章接下来的内容要解决的问题与 ReLU 有关，什么样的初始化方法可以最大化地发挥它的功效呢？

## 6.2 Xavier 初始化

6.1 节的结尾留下了一个疑问：什么样的初始化方法可以最大化发挥它的功效呢？所谓解铃还需系铃人，回答这个问题还要回到 6.1 节的第一个实验——那个结果优异的实验。当时模型的初始化方法是什么呢？**Xavier 方法**<sup>[1]</sup>。本节就详细介绍这个初始化方法的由来。

Xavier 的初始化算法如下。

定义参数所在层的输入维度为  $n$ ，输出维度为  $m$ ，那么参数将从处于  $\left[-\sqrt{\frac{6}{m+n}}, \sqrt{\frac{6}{m+n}}\right]$  范围内的均匀分布内进行采样初始化。

这个公式是如何计算出来的呢？想要回答这个问题，一段漫长的数学推导是不可避免的。作为一个理论推导，必要的假设不可或缺。Xavier 的推导过程主要基于以下三个假设。

1. 忽略偏置项对网络的影响。
2. 所有的非线性函数均为双曲正切函数 Tanh，且非线性函数的前向后向计算都近似为线性计算，因此它的影响也可以忽略。
3. 输入数据和参数相互独立。

第 1 个假设比较好理解，那么第 2 个假设是什么意思呢？Xavier 方法在创建时并没有把非线性函数假想成 ReLU，公式推导使用的非线性函数是 Tanh（但是从实际效果看，Xavier 和 ReLU 合作得很默契），在作者的设想中， $x$  和  $w$  都是靠近 0 且比较小的数字，那么它们最终计算出来的数字也应该是一个靠近 0，比较小的数字。那么再看一眼 Tanh 函数和它对应的梯度函数，如图 6-10 和图 6-11 所示。

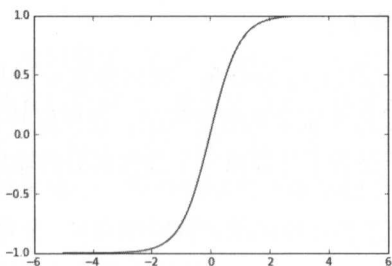


图 6-10 Tanh 函数的图像

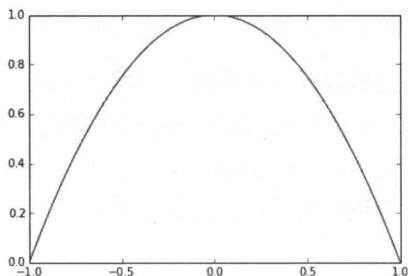


图 6-11 Tanh 函数的梯度图像

如果 Tanh 的输入数据在 0 附近，由于 Tanh 函数在以 0 为中心的领域导数接近 1，那么经过前向计算，输入和输出基本相同。所以作者告诉大家，基于这些原因 Tanh 在前向计算时可以忽略；同样地，因为靠近 0 的地方导数接近 1，那么后向计算时 Tanh 的梯度也几乎不影响参数的梯度求解。所以作者又一次告诉大家，在后向计算时也可以把 Tanh 忽略。

总之就是一句话：把它想象成一个可以忽略的线性函数。

于是这个非线性函数硬生生掰成一个线性函数，不过为了理论的完美这点牺牲也算不了什么。

在第 3 条假设中，参数和数据相互独立，实际上做到这一点也是基本不可能的。这里的假设同样是为了理论的完美。同时下面的推导将用到以下和方差相关的定理。

假设有随机变量  $x$  和  $w$ ，它们各自服从某个均值为 0，方差为  $\sigma$  的分布，那么：

- $w \cdot x$  就会服从均值为 0，方差为  $\sigma^2$  的分布
- $w \cdot x + w \cdot x$  就会服从均值为 0，方差为  $2 \cdot \sigma^2$  的分布

前面定理的变量名称是不是有点熟悉？没错，它们正是下面将要提到的参数  $w$  和输入数据  $x$ 。完成了 3 项假设，下面就要探讨一个核心问题，梯度消失，数值不稳定这些问题都是深层网络独有的，浅层网络并没有这些困扰。那么，如何让深层网络的数值在训练过程中的表现像浅层网络一样？

如果读者对浅层模型有一些了解，脑中就会迅速回忆起曾经接触过的浅层模型——Logistic Regression、SVM 等。为了让它们的表现更好，所有的数据都会进行预处理：例如做白化处理，使特征的均值方差保持在一个合理的范围内，然后用浅层模型训练完成。现在到了深层模型，对于输入层数据的白化是可以做到的——减去均值，除以标准差。对于内部层次的数据来说，将它们也做白化就有点困难了。（当然，后来真的有人完成了这件事，还做得不错，那就是 Batch Normalization）既然有点困难，我们就需要分析随着运算进行，模型内部的数值变成了什么样子。

这里假设所有的输入数据  $x$  满足均值为 0，方差为  $\sigma_x$  的分布，再将参数  $w$  以均值为 0，方差为  $\sigma_w$  的方式进行初始化。假设第一层是大家常见的卷积层，卷积层共有  $n$  个参数（ $n = \text{channel} \times \text{kernel\_h} \times \text{kernel\_w}$ ），于是为了计算出一个线性部分的输出结果，卷积层的计算公式如下所示：

$$z_j = \sum_i^n w_i \cdot x_i$$

其中  $j$  表示输出数据的下标， $i$  表示输入数据的下标。从公式中可以看出，线性部分的一个输出值，实际上是由  $n$  个输入值和参数值乘加计算出来的，那么按照前面对  $x$  和  $w$  的定义，和两个随机变量的方差计算公式，这个  $z$  会服从一个什么分布呢？

- 均值依然为 0。
- 方差变得复杂起来： $n \cdot \sigma_x \cdot \sigma_w$ 。

完成计算的数据将通过可以忽略的具有“线性特征”的非线性层，数值没有发生变化，那么下一层的数据就成了均值为 0，方差为  $n \cdot \sigma_x \cdot \sigma_w$  的“随机变量”产生的样本。

为了更好地表达，下面的公式中模型层号会写在变量的上标处。可以看出由于每一层的参数都是按照均值为 0 的分布进行初始化，那么下面各层输出数据的期望均值

也应该是 0，后面就不再单独分析它了。按照上面的推导过程，代表第 2 层网络输出的随机变量的方差将变成：

$$\sigma_x^2 = n^1 \cdot \sigma_x^1 \cdot \sigma_w^1$$

实际上每一层的方差计算是相同的，于是接着就可以计算代表第 3 层网络输出的随机变量的方差数值：

$$\sigma_x^3 = n^2 \cdot \sigma_x^2 \cdot \sigma_w^2$$

如果模型是一个  $k$  层的网络（这里主要卷指积层 + 全连接层的总和数），那么代表第  $k$  层网络输出的随机变量的方差就变成了：

$$\sigma_x^k = n^{k-1} \cdot \sigma_x^{k-1} \cdot \sigma_w^{k-1} = n^{k-1} \cdot n^{k-2} \cdot \sigma_x^{k-2} \cdot \sigma_w^{k-2} \cdot \sigma_w^{k-1}$$

继续把这个公式完全展开，最终的方差就变成了：

$$\sigma_x^k = \sigma_x^1 \cdot \prod_{i=1}^{k-1} n^i \cdot \sigma_w^i$$

可以看出，等式右边的那个连乘项就像一个定时炸弹，如果  $n^i \cdot \sigma_w^i$  总是大于 1，那么随着层数越深，数值的方差会越来越大，数值的幅度也会越来越大，这样就有可能造成 6.1 节实验 2 的情况；反过来，如果乘积小于 1，那么随着层数越深，数值的方差就会越来越小，模型的输出会逐渐趋于统一，这样又有可能发生第 3 章中神经网络退化的情况。这样的问题浅层网络不会遇到，而深层模型却无法回避，如果处理不好，深层模型的表现就会很不好。

那么，有没有什么办法解决方差的问题呢？这里不妨回头看看这个公式：

$$\sigma_x^2 = n^1 \cdot \sigma_x^1 \cdot \sigma_w^1$$

读者一定会有这样一个想法，如果  $\sigma_x^2 = \sigma_x^1$ ，接着保证每一层输入数据的方差都保持相同，那么数值的幅度不就可以控制住了吗？于是上面的公式经过变换，可以得到：

$$\sigma_w^1 = \frac{1}{n^1}$$

这样看来，只要用均值为 0，方差为  $\frac{1}{n}$  的分布对第一层网络参数初始化，再用同样的方法对其他层的参数进行初始化，不就可以解决问题了？从理论上讲这个方法是正确的，但这个方法不够完善，前面的推导只关注了前向计算的角度，那么后向计算呢？

后向的计算公式其实和前向类似，对于前面假设的  $k$  层的网络，现在得到了第  $k$  层的梯度  $\frac{\partial \text{Loss}}{\partial x^k}$ ，那么对于第  $k-1$  层输入数据的梯度，有：

$$\frac{\partial \text{Loss}}{\partial x_j^{k-1}} = \sum_{i=1}^{\hat{n}} \frac{\partial \text{Loss}}{\partial x_i^k} \cdot w_j^k$$

也就是说， $k-1$  层输入数值的梯度，相当于第  $k$  层输入数据的梯度和本层参数的乘加。由于是反向计算，这里的  $\hat{n}$  和前面的  $n$  表示的维度不同。

根据前向计算中的推导，如果用  $\nabla x_j^k$  表示代表了第  $k$  层第  $j$  个元素梯度的随机变量，那么和前向推导类似的公式就可以写出来了：

$$\text{Var}(\nabla x_j^{k-1}) = \hat{n}^k \cdot \text{Var}(\nabla x_i^k) \cdot \sigma_w^k$$

将公式中的变量展开就可以推导出包含连乘的公式：

$$\text{Var}(\nabla x_j^1) = \text{Var}(\nabla x_i^k) \cdot \prod_{i=1}^{k-1} \hat{n}^i \cdot \sigma_w^i$$

前面提到了连乘对前向计算可能造成的影响，那么上面公式的连乘对反向传播的影响也是很大的。因此，除了确保前向计算的数值稳定，反向计算的稳定也十分必要。如果反向计算不能做到同样的数值稳定，那么被不稳定的更新量更新过的数值不再服前面假设的分布，整个模型的数值稳定性依然无法保证。

所以为了确保稳定，每一层的梯度的方差也要保持相同，也就是说：

$$\text{Var}(\nabla x_j^{k-1}) = \text{Var}(\nabla x_i^k)$$

由此得到：

$$\sigma_w^k = \frac{1}{\hat{n}^k}$$

看上去前后向计算推导得到了同样的结果，但如果读者仔细看这两个公式，就会发现两个表示参与运算数量的  $n$  实际上不是同一个  $n$ 。对于全连接来说，前向操作时， $n$  表示了输入的维度，而后向操作时， $\hat{n}$  表示了输出的维度。于是作者把两个公式揉合在一起，最终变成：

$$\sigma_w^k = \frac{2}{\hat{n}^k + n^k}$$

到此，参数方差的推导终于完成了，下面的工作就是设计一个用于初始化参数的概率分布。在论文中，作者使用均匀分布进行参数初始化。熟悉和不熟悉均匀分布的各

位读者，都可以一起回顾均匀分布的方差计算公式。假设随机变量服从范围为  $[a, b]$  的均匀分布，那么它的方差为：

$$\sigma = \frac{(b-a)^2}{12}$$

这里假设参数初始化的范围是  $[-a, a]$ 。把上面的公式套进去，就可以求出参数初始化中的  $a$ ：

$$\text{Var} = \frac{(a - (-a))^2}{12} = \frac{a^2}{3} = \sigma_w^k$$

将  $\sigma_w^k$  的结果带入，就可以得到：

$$a = \sqrt{\frac{6}{\hat{n}^k + n^k}}$$

于是，Xavier初始化方法终于推导完成，那就是把参数初始化成  $[-\sqrt{\frac{6}{\hat{n}^k + n^k}}, \sqrt{\frac{6}{\hat{n}^k + n^k}}]$  范围内的均匀分布。以下是 Caffe 中 Xavier 方法初始化参数的代码，其中与核心功能无关的代码已经被去掉：

```
virtual void Fill(Blob<Dtype>* blob) {
    int fan_in = blob->count() / blob->num();
    int fan_out = blob->count() / blob->channels();
    Dtype n = (fan_in + fan_out) / Dtype(2);
    Dtype scale = sqrt(Dtype(3) / n);
    caffe_rng_uniform<Dtype>(blob->count(), -scale, scale,
        blob->mutable_cpu_data());
}
```

看完了上面大段的推导，再看看最终的算法和源代码，有没有一种把简单问题复杂化了的感觉？没错，这个初始化公式的最终形态并不难，但是这个公式背后的思想和所用到的抽象思维是十分巧妙的，从复杂而严谨的理论到简洁实用的实现，作者为此付出了巨大的心血。读者在使用这个初始化方法时，会发现这个方法在实战中很好用。

最后还要强调一点，虽然前面用了一些戏谑的语言来介绍推演中的假设，并不代表本书对假设的否定。如果没有这些假设，这样精彩而且实用的结论就很难得出。其实在数学模型的世界中，抽象、假设都是经常用到的工具，只有把一些难以掌控的地方去掉，才更容易从宏观上抓住事物的本质，找到事物的核心规律。所以这里还是要由衷地为 Xavier 初始化算法的作者点赞。当然，换个角度看，充满假设的模型毕竟不够精确，想要把数值问题解决得更好，我们仍然需要继续努力，寻找更精确的模型。

### 6.3 MSRA 初始化

**MSRAFiller**<sup>[2]</sup> 是 Caffe 中实现的另一个知名参数初始化方法，它的公式推导思路和 Xavier 类似，但是前提假设略有不同。6.2 节介绍的 Xavier 方法的假设中，模型的非线性函数是可以忽略的，这对 Tanh 这样的非线性函数是可以近似的，但是如果换成 ReLU 的话，这样的假设恐怕有些说不过去。MSRA 初始化法主要就是解决这个问题，作者希望在基于非线性层为 ReLU 和 PReLU 的基础上再做一次针对参数数值和梯度方差的推导，从而得出一个新的参数初始化方法。

#### 6.3.1 前向推导

首先定义推导过程中需要的一些变量，卷积操作可以写成下面这个公式，这和第 4 章介绍的“偶像派”解法的形式类似，输入数据变成向量，卷积核组成矩阵。

$$y_l = W_l x_l + b_l$$

其中  $x$  的维度为  $n = c \times k \times k$ ， $c$  代表输入数据的通道数目， $k$  是卷积核的维度，这里我们假设卷积核的长宽相同。 $W$  的维度为  $d \times n$ ，其中  $d$  为输出的通道数目。 $b$  是偏置项，它的维度为  $d$ 。 $l$  表示了当前所在层的层号，这里假设模型中没有 stride>1 的场景，没有 pooling 层，那么  $c_l = d_{l-1}$ ，上面公式的关系也可以用图 6-12 表示。

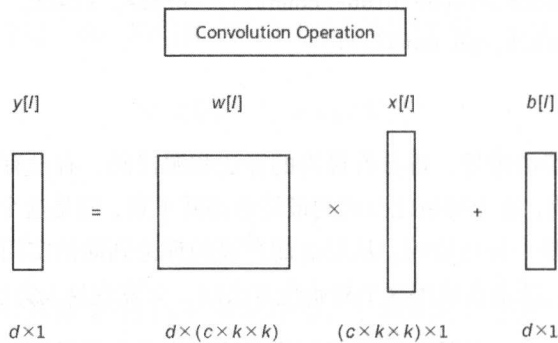


图 6-12 卷积操作的局部尺度信息图

这里假设  $x$  的每个元素都是独立同分布的。同时  $W$  的每一个元素也是独立同分布的，均值也为 0。那么计算方差的公式就可以得出来：

$$\text{Var}[y_l] = n_l \text{Var}[w_l x_l]$$

如果令  $w_l$  的期望  $E[w_l]$  为 0, 那么公式还可以做进一步变换, 利用方差公式:  $\text{Var}[x] = E[x^2] - E[x]^2$ , 可以得到:

$$\begin{aligned}
 \text{Var}[y_l] &= n_l \text{Var}[w_l x_l] \\
 &= n_l (E[w_l^2 x_l^2] - E[w_l x_l]^2) \\
 &= n_l (E[w_l^2] E[x_l^2] - (E[w_l] E[x_l])^2) \\
 &= n_l ((\text{Var}[w_l] + E[w_l]^2) E[x_l^2]) \\
 &= n_l \text{Var}[w_l] E[x_l^2]
 \end{aligned}$$

由于 ReLU 激活函数的存在, 线性部分的输出不具有正负对称性, 所以  $x$  的期望并不等于 0,  $E[x_l^2]$  和  $\text{Var}[y_{l-1}]$  并不相同。从这里开始, MSRA 方法就和 Xavier 不同了。下面需要想办法把  $E[x_l^2]$  求出来。

然而,  $E[x_l^2]$  也不是那么直观的, 需要我们从经典的分布形式归纳得出来。这里假设  $w_{l-1}$  被初始化得非常均匀, 而且分布关于 0 对称, 线性部分计算得到的  $y_{l-1}$  也非常均匀地对称。如果假设  $y_{l-1}$  服从均匀分布且上下界关于 0 对称, 令它的上界为  $k$ , 那么就有:

$$\text{Var}[y_{l-1}] = \frac{(k - (-k))^2}{12} = \frac{k^2}{3}$$

经过 ReLU 层后,  $x_l$  的数据有一半变成了 0, 另一半变成了从 0 到  $k$  的均匀分布, 那么就有:

$$\begin{aligned}
 E[x_l] &= 0 \times \frac{1}{2} + \frac{k}{2} \times \frac{1}{2} = \frac{k}{4} \\
 E[x_l]^2 &= \frac{k^2}{16} \\
 \text{Var}[x_l] &= \frac{1}{2} \times (0 - \frac{k}{4})^2 + \frac{1}{2} \frac{\int_0^k (x - \frac{k}{4})^2 dx}{k} \\
 &= \frac{k^2}{32} + \frac{1}{2} \frac{\int_0^k (x^2 - \frac{k}{2}x + \frac{k^2}{16}) dx}{k} \\
 &= \frac{k^2}{32} + \frac{1}{2} \frac{(\frac{1}{3}x^3 - \frac{k}{4}x^2 + \frac{k^2}{16}x)|_0^k}{k} \\
 &= \frac{k^2}{32} + \frac{1}{2} (\frac{1}{3}k^2 - \frac{1}{4}k^2 + \frac{1}{16}k^2) \\
 &= \frac{k^2}{32} + \frac{7}{96}k^2 \\
 &= \frac{10}{96}k^2
 \end{aligned}$$



所以有：

$$E(x_l^2) = E(x_l)^2 + \text{Var}(x_l) = \frac{1}{16}k^2 + \frac{10}{96}k^2 = \frac{1}{6}k^2 = \frac{1}{2}\text{Var}(y_{l-1})$$

基于均匀分布的假设，两者的关系被求了出来。那么，别的分布会不会有同样的答案呢？比方说常见的正态分布？由于正态分布的公式推导太过复杂且非常不实用，这里就不采用公式推导的方法解决，转而使用采样的方法验证这个推导的正确性，代码如下：

```
import numpy as np
y = np.random.normal(0.0,1.0,100000000)
var_y = np.var(y)
x = y
x[x < 0] = 0
mean_x = np.mean(x)
var_x = np.var(x)
mean_x_square = mean_x ** 2 + var_x
print str(abs(var_y / 2 - mean_x_square))
```

经过一段时间的计算，我们得到了下面的结果：

3.500e-05

可以看出，我们的结果在大规模数据采样下是站得住脚的，那就使用上面的结论继续推导：

$$E(x_l^2) = \frac{1}{2}\text{Var}(y_{l-1})$$

将前面的公式合起来，就得到了下面这条递推公式：

$$\text{Var}[y_l] = \frac{1}{2}n_l\text{Var}[w_l]\text{Var}[y_{l-1}]$$

这样就完成了递推公式的推导，和 6.2 节的方法类似，把多层的公式集合到一起，就可以得到：

$$\text{Var}[y_L] = \text{Var}[y_1](\prod_{l=2}^L \frac{1}{2}n_l\text{Var}[w_l])$$

到此为止，有了前面推导 Xavier 公式的经验，下面的步骤就很清晰了。为了保证

数值幅度的稳定性，不同层次的输出值的方差需要保持相同，于是有：

$$\frac{1}{2}n_l \text{Var}[w_l] = 1$$

### 6.3.2 后向推导

完成了前向推导，下面再来看看后向推导。后向推导的关键公式如下所示：

$$\Delta x_l = \hat{W}_l \Delta y_l$$

其中  $\Delta y_l$  表示损失函数对第  $l$  层输出的偏导，它的维度是  $\hat{n} = d \times k \times k$ ，表示  $d$  个通道的  $k \times k$  个卷积核所影响的范围。 $\Delta x_l$  表示损失函数对第  $l$  层输入的偏导，它的维度是  $c \times 1$ 。根据前面的假设， $\hat{W}_l$  的参数由均值为 0 且分布形状以  $x = 0$  处对称的分布初始而成，那么  $E[\Delta x_l] = 0$ ，另一个公式是：

$$\Delta y_l = f'(y_l) \cdot \Delta x_{l+1}$$

这里的导数是非线性部分 ReLU 的导数，它的值非 0 即 1。接着就来计算反向传播方差的递推公式。如果在保持前面关于  $y$  的分布的假设基础上，继续假设  $f'(y_l)$  和  $\Delta x_{l+1}$  之间相互独立，那么就有：

$$\begin{aligned} E[\Delta y_l] &= E[f'(y_l)]E[\Delta x_{l+1}] = \frac{1}{2}E[\Delta x_{l+1}] = 0 \\ \text{Var}[\Delta y_l] &= E[(\Delta y_l)^2] - E[\Delta y_l]^2 \\ &= E[(\Delta y_l)^2] \\ &= E[f'(y_l)^2 (\Delta x_{l+1})^2] \\ &= E[f'(y_l)^2]E[(\Delta x_{l+1})^2] \\ &= E[f'(y_l)^2](E[(\Delta x_{l+1})^2] + \text{Var}(\Delta x_{l+1})) \\ &= E[f'(y_l)^2]\text{Var}(\Delta x_{l+1}) \\ &= \frac{1}{2}\text{Var}(\Delta x_{l+1}) \end{aligned}$$

有了上面的结论，我们就得到了这个推导公式：

$$\text{Var}[\Delta x_l] = \frac{1}{2}\hat{n}_l \text{Var}[w_l] \text{Var}[\Delta x_{l+1}]$$

同样，把  $L$  个模型层的参数展开，就可以得到：

$$\text{Var}[\Delta x_2] = \text{Var}[\Delta x_{L+1}] \left( \prod_{l=2}^L \frac{1}{2} \hat{n}_l \text{Var}[w_l] \right)$$

为了满足梯度的数值稳定性，这里同样需要保证：

$$\frac{1}{2} \hat{n}_l \text{Var}[w_l] = 1$$

这样得到了前向后向两个方向的计算公式，下面的步骤同 Xavier 算法一样，将两部分得到的结论融合起来，就得到了最终的初始化算法。作者采用高斯分布进行初始化，分布的均值为 0，方差为  $\sqrt{\frac{4}{n_l + \hat{n}_l}}$ 。下面是 Caffe 代码的缩略版：

```
virtual void Fill(Blob<Dtype>* blob) {
    int fan_in = blob->count() / blob->num();
    int fan_out = blob->count() / blob->channels();
    Dtype n = (fan_in + fan_out) / Dtype(2);
    Dtype std = sqrt(Dtype(2) / n);
    caffe_rng_gaussian<Dtype>(blob->count(), Dtype(0), std,
        blob->mutable_cpu_data());
}
```

到此 MSRA 初始化方法也介绍完了，那么在设计模型时应该选择哪一种呢？这需要针对具体问题多做尝试才行。还是那句话，了解两种参数初始化方法的关键是理解方法背后对模型计算抽象和分析的方法。

## 6.4 ZCA

前面介绍了许多参数初始化的方法，本节将介绍输入数据的初始化方法。相信读者也了解一些常用的初始化方法：例如保证一批数据的均值为 0，方差为 1 的预处理。第 5 章也提到，在 Caffe 的网络描述中，Data Transformer 的配置中有一项是配置 mean\_file 的路径，也就是存储数据的平均值的文件。在训练时，每个数据在进入网络计算前都会减去平均值，这样可以确保训练数据的整体均值为 0，也就和 6.2 节、6.3 节中对输入数据的假设相近。那么除了上面提到的方法之外，还有其他初始化的方法吗？本节将介绍 ZCA（Zero Component Analysis）这个经典的初始化算法。

本节将用一个例子讲述这个初始化算法的过程。首先利用随机算法生成一个 2 维的数据集，数据集的两个维度存在强相关的关系。生成数据的代码如下所示：

```
x = np.random.randn(200)
y = x * 2
err = np.random.rand(200) * 2
y += err
data = np.vstack((x,y))
plt.scatter(data[0,:], data[1,:])
```

数据图像如图 6-13 所示。

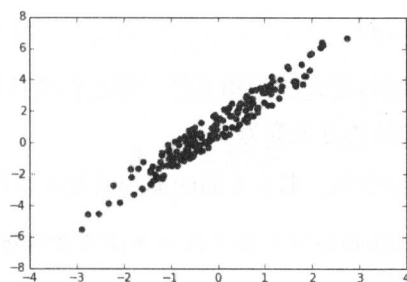


图 6-13 相关特征的 2 维示例数据

在进行变换前，需要求出两个维度的均值，再让全体数据减去均值，使得整体数据的均值为 0：

```
mean = np.mean(data, axis=1)
data -= mean.reshape((mean.shape[0],1))
plt.scatter(data[0,:], data[1,:])
```

结果如图 6-14 所示。

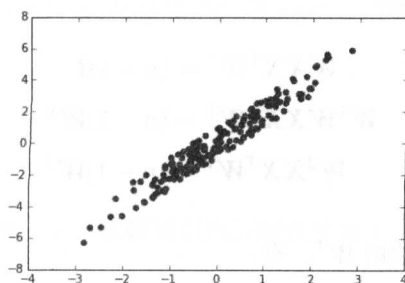


图 6-14 使均值为 0 后的数据

下面就是 ZCA 的关键部分。读者应该知道训练数据中有时会出现特征之间相互关联的问题，对于图像数据，相互关联的问题更严重。虽然卷积层可以通过学习来解决这些局部相关性，但是把这个事情交给模型学习总是不够直接，如果这个问题能够在输入数据这个环节得到处理，那么训练一定会更容易些。

接下来计算数据的协方差，由于现在数据的均值为 0，计算协方差就简化成了如下的计算：

```
conv = np.dot(data, data.T) / (data.shape[1] - 1)
print conv
#以下为结果显示
[[ 0.88200859  1.80316947]
 [ 1.80316947  4.033871  ]]
```

ZCA 变换的目标是去除特征之间的相关性，那么它的目标就是：让每个特征自身的方差变为 1，让特征之间的协方差变为 0。

ZCA 的主要手段是线性变换，那么上面的那句话就可以转换成下面形式化的表达：

```
# 如果有数据矩阵X，那么ZCA的目标是要寻找一个线性变换矩阵W，满足
Y = np.dot(W, X)
# 且
np.dot(Y, Y.T) / (Y.shape[1] - 1) == np.eye(Y.shape[0])
```

为了达到这个目标，ZCA 算法首先做了如下假设：

线性变换矩阵  $W$  是一个对称矩阵： $W = W^T$ ，从 2.2 节的内容已经知道，对称矩阵具有很好的性质，这个性质对后面的公式推导很有用。

由于目标是令

$$YY^T = (n - 1)I$$

那么公式就可以转变为：

$$\begin{aligned} WXX^TW^T &= (n - 1)I \\ W^TWXX^TW^T &= (n - 1)W^T \\ W^2XX^TW^T &= (n - 1)W^T \end{aligned}$$

同时去掉左右两式右边的  $W^T$ ，有：

$$W^2XX^T = (n - 1)I$$

假设  $\mathbf{X}$  不全为 0，那么  $\mathbf{X}\mathbf{X}^T$  就是一个对称矩阵，满足可逆性，所以：

$$\begin{aligned} \mathbf{W}^2 &= (n-1)(\mathbf{X}\mathbf{X}^T)^{-1} \\ \mathbf{W} &= \sqrt{n-1}(\mathbf{X}\mathbf{X}^T)^{-1/2} \end{aligned}$$

到此为止，实际上 ZCA 的计算公式推导已经告一段落，下面要做的是解决公式中对矩阵开根号的计算问题。由于  $\mathbf{X}\mathbf{X}^T$  是一个对称矩阵，对称矩阵可以被对角化。首先求出  $\mathbf{X}\mathbf{X}^T$  的特征值和特征向量：

$$\mathbf{X}\mathbf{X}^T\mathbf{S} = \mathbf{S}\mathbf{\Lambda}$$

前面提到对称矩阵具有一个特性，它的特征向量可以构成一个标准正交矩阵，根据标准正交矩阵的特性——它的逆矩阵和转置矩阵相等，于是可以得到：

$$\mathbf{X}\mathbf{X}^T = \mathbf{S}\mathbf{\Lambda}\mathbf{S}^T$$

继续推导，根据 2.2 节可以得到：

$$(\mathbf{X}\mathbf{X}^T)^{-1/2} = (\mathbf{S}\mathbf{\Lambda}\mathbf{S})^{-1/2} = \mathbf{S}\mathbf{\Lambda}^{-1/2}\mathbf{S}$$

于是最终的

$$\mathbf{W} = \sqrt{n-1}\mathbf{S}\mathbf{\Lambda}^{-1/2}\mathbf{S}$$

以下是推导对应的代码：

```
# 由于conv中已经除掉了1/(Y.shape[1] - 1),所以后面的计算中我们将不再去除它
eig_val, eig_vec = np.linalg.eig(conv)
S_sqrt = np.sqrt(np.diag(eig_val))
W = np.dot(eig_vec, np.dot(np.linalg.inv(S_sqrt), eig_vec.T))
print W
#以下为结果显示
[[ 3.376 -1.327]
 [-1.327  1.056]]
```

现在  $\mathbf{W}$  已经求解出来，下一步就可以进行线性变换了：

```
Y = np.dot(W, data)
plt.scatter(Y[0,:], Y[1,:])
```

```
conv2 = np.dot(Y, Y.T) / (data.shape[1] - 1)
print conv2
```

对应的结果如图 6-15 所示。

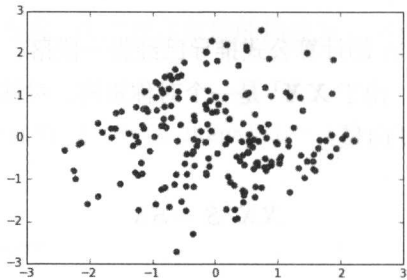


图 6-15 ZCA 处理的结果

此时经过转换后的数据对应的协方差为：

```
[[ 1.000e+00 -1.294e-16]
 [ -1.294e-16  1.000e+00]]
```

无论从图像结果还是协方差的数值结果上看，ZCA 都完成了目标。以上的 ZCA 算法推导来自论文 *Learning Multiple Layers of Features from Tiny Images*<sup>[3]</sup> 的附录，感兴趣的读者可以阅读原文寻找更多灵感。另外，ZCA 初始化在一些经典的数据集（比方说 CIFAR 10）上已经得到验证，采用这样的初始化可以提高最终的识别精度。读者朋友不妨动手一试。

## 6.5 与数值溢出的战斗

和前面的内容不同，本节主要介绍计算 Loss Layer 过程中可能出现的一些数值问题及解决方法。这部分和理论知识关系不大，主要是实战中解决由于计算机系统自身存在缺陷所造成的问题。本节的研究对象是 Caffe 的 Softmax Layer 和 Sigmoid Cross Entropy Loss Layer 两个模型层，通过研究 Caffe 中的具体实现来了解这类数值问题的解决方法。

### 6.5.1 Softmax Layer

Softmax Layer 中的 Softmax 函数是一种泛化的 Logistic 函数，它将一个  $N$  维的实数向量压缩成一个满足特定条件的  $N$  维实数向量。压缩后的向量满足两个条件。

1. 向量中每个元素的大小都在  $[0, 1]$ 。
2. 所有向量元素的和为 1。

因此，这个方法特别适合得到多分类问题对每一个类别的概率判断。它的计算公式如下所示：

$$\text{Softmax}(x_i) = \frac{e^{x_i}}{\sum_i e^{x_i}}$$

对应的 Python 代码如下所示：

```
def naive_softmax(x):
    y = np.exp(x)
    return y / np.sum(y)
```

为了展示函数的效果，下面随机生成一组数据，并显示出数据经过 Softmax 函数计算的结果：

```
a = np.random.rand(10)
print a
print naive_softmax(a)
#以下为结果显示
[ 0.673  0.203  0.020  0.299  0.231  0.439  0.982  0.545  0.003  0.834]
[ 0.122  0.076  0.063  0.084  0.078  0.096  0.166  0.107  0.062  0.143]
```

从结果来看数值表现正常，但是如果输入不那么正常——比方说数据很大呢？

```
b = np.random.rand(10) * 1000
print b
print naive_softmax(b)
#以下为结果显示
[ 497.467  227.753  537.826  787.549  663.138  224.693  958.394  139.096
 381.350  604.085]
[ 0.  0.  0. nan  0.  0. nan  0.  0.  0.]
```

从结果可以看出，数值溢出了，因为当今计算机的设计，常规数据类型在表示数字时存在上限，而指数函数的结果通常又很大，所以结果溢出是一件很常见的事情。那么输入大概到多少会溢出呢？

```
np.exp(709)
8.218e+307
```

这是在 Python 中能够正常输出的数字的最大正整数。实际上，这个结果已经很接近 double 类型能表示的数字上限了。



虽然本章前面讲过有一些参数初始化的方法可以控制住数值的范围，使输出不会那么大，但难免会有比较大的计算结果。实际场景中计算 Softmax 结果的向量维度可能很大，那么比较大的数字积累起来，就很有可能造成数值溢出的结果。

这个问题如何解决呢？解决的方法是做除法，只要同时给分子分母除以一个大数，使得最终的结果不溢出，问题就可以解决。一般来说，为了保证结果不溢出，首先选出输入数据中最大的数，然后让分子分母除以 e 的最大数次幂：

$$\text{Softmax}(x_i) = \frac{e^{x_i}}{\sum_i e^{x_i}} = \frac{\frac{e^{x_i}}{e^{\max_x}}}{\sum_i \frac{e^{x_i}}{e^{\max_x}}} = \frac{e^{x_i - \max_x}}{\sum_i e^{x_i - \max_x}}$$

当然，指数除法可以转化为减法，于是代码可以写成这样：

```
def high_level_softmax(x):
    max_val = np.max(x)
    x -= max_val
    return naive_softmax(x)
```

经过这样的变换，之前溢出的问题就得到了解决：

```
b = np.random.rand(10) * 1000
print b
print high_level_softmax(b)
#以下为结果显示
[ 903.274  260.683   22.316  544.806  506.268  698.380  833.720  200.556
 924.077  909.398]

[ 9.233e-010  7.790e-289  0.000e+000  1.925e-165  3.531e-182  9.570e-099
 5.733e-040  6.011e-315  9.999e-001  4.217e-007]
```

溢出的问题得到了解决，但是 Softmax 的结果恐怕和读者的直觉有些偏差。上面的例子中最大的输入数字 924.077 的计算结果高达 0.99，而其他一众数字经过 Softmax 计算之后都小得可怜，小到我们用肉眼无法从坐标轴上把它们区分出来，即使和最大数字相差不大的最后一个输入数字 909.398，经过计算后都变成了一个非常小的数字  $4.2 \times 10^{-7}$ ，它与最大数的结果相差数万倍。这说明 Softmax 的最终结果和数值大小有很大的关系。如果数据的跨度比较大，那么 Softmax 并不能很好地刻画原始数据之间的关系。为了让这些小得可怜的数字不那么可怜，使用一点平滑的小技巧是很有必要的，于是代码又变成：

```
def practical_softmax(x):
```

```

max_val = np.max(x)
x -= max_val
y = np.exp(x)
y[y < 1e-20] = 1e-20
return y / np.sum(y)
#以下为结果显示
[ 9.233e-10  9.999e-21  9.999e-21  9.999e-21  9.999e-21  9.999e-21
 9.999e-21  9.999e-21  9.999e-01  4.217e-07]

```

看上去比上面的结果稍好一些，但依然不能扭转一家独大的局面。想要从根本上解决问题，要么换掉 Softmax 函数，要么控制数据的维度，这两件事情总要去尝试一件。

### 6.5.2 Sigmoid Cross Entropy Loss

从上面的例子可以看出，exp 函数是一个“危险”函数。下面又轮到另外一个容易出现数值溢出问题的函数——Sigmoid 出场了。下面的主角是 Sigmoid Cross Entropy Loss 函数，曾经在 2.8 节中介绍过这个函数，它的代码如下所示：

```

def naive_sigmoid_loss(x, t):
    y = 1 / (1 + np.exp(-x))
    return -np.sum(t * np.log(y) + (1 - t) * np.log(1 - y)) / y.shape[0]

```

首先给一个温和的例子，这里先随机生成一批模型预测结果的数据，再生成一批表示正确结果的数据，然后用上面的函数计算，并显示最终结果：

```

a = np.random.rand(10)
b = a > 0.5
print a
print b
print naive_sigmoid_loss(a,b)
#以下为结果显示
[ 0.399  0.043  0.186  0.054  0.827  0.163  0.185  0.574  0.630  0.627]
[False False False False  True False False  True  True  True]
0.637

```

在温和的例子中，函数表现正常，而在下面的例子中，数值问题将再一次出现：

```
a = np.random.rand(10)* 1000
b = a > 500
print a
print b
print naive_sigmoid_loss(a,b)
#以下为结果显示
[ 63.208  958.943  250.753  895.493  965.626  81.121  423.364  532.206
 333.454  185.726]
[False  True False  True  True False False  True False False]
nan
```

果然不出所料，程序又一次溢出了。这一次解决的方式同前面类似，公式需要进行一些转换，避免指数运算时数字太大。

首先输入数据经过 Sigmoid 函数后输出为 1 的概率为：

$$\hat{p}_n = \frac{1}{1 + e^{-x_n}}$$

将这个公式代入 Cross-Entropy loss 中，可以得到：

$$\begin{aligned} \text{Loss} &= -\frac{1}{N} \sum_{n=1}^N [p_n \log(\hat{p}_n) + (1 - p_n) \log(1 - \hat{p}_n)] \\ &= -\frac{1}{N} \sum_{n=1}^N [p_n \log(\frac{1}{1 + e^{-x_n}}) + (1 - p_n) \log(1 - \frac{1}{1 + e^{-x_n}})] \\ &= -\frac{1}{N} \sum_{n=1}^N [p_n \log(\frac{1}{1 + e^{-x_n}}) + (1 - p_n) \log(\frac{e^{-x_n}}{1 + e^{-x_n}})] \\ &= -\frac{1}{N} \sum_{n=1}^N [p_n \log(\frac{1}{1 + e^{-x_n}}) - p_n \log(\frac{e^{-x_n}}{1 + e^{-x_n}}) + \log(\frac{e^{-x_n}}{1 + e^{-x_n}})] \\ &= -\frac{1}{N} \sum_{n=1}^N [p_n \log(\frac{1}{1 + e^{-x_n}} \cdot \frac{1 + e^{-x_n}}{e^{-x_n}}) + \log(\frac{e^{-x_n}}{1 + e^{-x_n}})] \\ &= -\frac{1}{N} \sum_{n=1}^N [p_n x_n + \log(\frac{e^{-x_n}}{1 + e^{-x_n}})] \end{aligned}$$

这个 Loss 函数中包含指数运算，为了防止溢出，这里需要保证指数的幂不大于 0，于是公式需要分两种情况考虑。

当  $x_n \geq 0$ , 指数项将保持不变, 即  $\log(\frac{e^{-x_n}}{1+e^{-x_n}})$ , 于是公式就变成:

$$\begin{aligned} &= -\frac{1}{N} \sum_{n=1}^N [p_n x_n + \log(e^{-x_n}) - \log(1 + e^{-x_n})] \\ &= -\frac{1}{N} \sum_{n=1}^N [(p_n - 1)x_n - \log(1 + e^{-x_n})] \end{aligned}$$

当  $x_n < 0$ , 指数项将被反转为:  $\log(\frac{1}{1+e^{x_n}})$ , 于是公式就变成:

$$= -\frac{1}{N} \sum_{n=1}^N [p_n x_n - \log(1 + e^{x_n})]$$

于是, 代码变成了:

```
def high_level_sigmoid_loss(x, t):
    first = (t - (x > 0)) * x
    second = np.log(1 + np.exp(x - 2 * x * (x > 0)))
    return -np.sum(first - second) / x.shape[0]
```

这时再来看看前面的极端例子:

```
a = np.random.rand(10)* 1000 - 500
b = a > 0
print a
print b
print high_level_sigmoid_loss(a,b)
#以下为结果显示
[-173.48716596  462.06216262 -417.78666769    6.10480948  340.13986055
 23.64615392  256.33358957 -332.46689674  416.88593348 -246.51402684]
[False  True False  True  True  True  True False  True False]
0.000222961919658
```

这样一来数值的问题就解决了。由于当前计算机体系的设计问题, 像指数计算这样的溢出问题短时间内仍然无法彻底解决, 因此在计算中每当遇到指数计算, 读者都需要想办法进行变换, 从而保证数值溢出问题不会发生。

## 6.6 总结

本章完成了对深层网络数值相关的分析与讨论，让我们来回顾一下。

- ReLU 宽广的接受域使得它对参数的数值范围比较敏感。
- Xavier 和 MSRA 方法都是优秀的参数初始化方法，使用它们可以很好地解决数值范围不稳定的问题。
- ZCA 是一种打破特征相关性的初始化方法。
- 指数运算存在溢出的风险，为了避免溢出，指数运算需要被合理转化。

## 6.7 参考文献

[1] Glorot X, Bengio Y. Understanding the difficulty of training deep feedforward neural networks[J]. Journal of Machine Learning Research, 2010, 9:249-256.

[2] He K, Zhang X, Ren S, et al. Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification[J]. 2015:1026-1034.

[3] Krizhevsky A. Learning Multiple Layers of Features from Tiny Images[J]. 2012.

# 7

## 网络结构

本章介绍 CNN 网络架构方面的内容。网络架构也是 CNN 的一个核心部分，由于网络层数深，深度模型的网络架构给了人们很大的发挥空间，于是有无数的科研人员创造了各种各样的模型。本章将介绍对 CNN 发展产生巨大影响的部分模型结构，以及在网络中起特殊作用的模型层，最后通过实验分析模型层在网络结构中起到的作用。

### 7.1 关于网络结构，我们更关心什么

本节将回顾 CNN 发展过程中的部分经典模型结构，回顾的方式是从理性的角度计算与模型结构有关的一些具体数字，从而描绘出这些模型的一个简单轮廓。这一回的目标问题不再是 MNIST，而是 ImageNet 数据集中的分类问题，这个数据集包含了 100 多万张自然场景拍摄的图像，其中包含了 1000 个类别。模型结构的关注点有下面几个。

1. 模型的总深度。这代表了模型的学习训练数据“能力”和模型的复杂度。基本上大家都有一个共识，一般情况下，越深的模型在函数拟合方面的能力越强。计算的方法是直接利用 Caffe 输出的 `layers_.size()` 的数值。由于其中还包括 Data Layer 和 Loss Layer，所以统计数会比实际的有效层数多。
2. 模型的参数总量。这一项同样代表了模型的学习能力和复杂度。从机器学习的理论上讲，参数越多，模型的表达能力也会“越强”。这里通过 Caffe 框架计算所有 `learnable_params` 的数量总和得到结果。
3. 模型前向计算所需的内存量。也就是 Caffe 中 `memory_used` 变量值的数量。

我们将要回顾的经典模型如下。

**AlexNet**<sup>[1]</sup>：2012 年令众人震惊的网络。模型的 prototxt 来自 Caffe 官方提供的模型描述文件：[https://github.com/BVLC/caffe/blob/master/models/bvlc\\_alexnet/train\\_val.prototxt](https://github.com/BVLC/caffe/blob/master/models/bvlc_alexnet/train_val.prototxt)。

**VGGNet**<sup>[2]</sup>：一个非常有代表性的网络，这个网络创建了一种自己的模型“哲学”。VGG19 层的模型的 prototxt 来自：<https://gist.github.com/ksimonyan/3785162f95cd2d5fee77#file-readme-md>。

**GoogLeNet**<sup>[3]</sup>：GoogLeNet 作为 Inception module 的实践者，曾经取得过非常不错的成绩，它的模型结构也很有特点。参考的 prototxt 来自：[https://github.com/BVLC/caffe/blob/master/models/bvlc\\_googlenet/train\\_val.prototxt](https://github.com/BVLC/caffe/blob/master/models/bvlc_googlenet/train_val.prototxt)。

**ResNet**<sup>[4]</sup>：ResNet 作为新一代的模型霸主，其对模型构建的思想可谓又上了一个台阶。这里的 ResNet 我们参考的 prototxt 是<https://github.com/KaimingHe/deep-residual-networks/blob/master/prototxt/ResNet-152-deploy.prototxt>。

表 7-1 给出了四个网络的统计结果，并附上论文中或者网络上给出的单模型的精度。

表 7-1 经典模型的基本信息

	层数	模型内存 (GB)	参数总量 (M)	Top1 Error	Top5 Error
AlexNet	24	2.12	60.96	39.0	16.6
VGGNet	46	33.44	175.12	25.5	8.0
GoogLeNet	169	1.7	13.37	31.3	11.1
ResNet	721	16.2	60.34	21.43	5.71

下面我们就按列进行分析。

- 从模型层数来看，几年间模型的层数已经得到了爆炸式的增长，虽然 GoogLeNet 的 Inception module 和 ResNet 的 Residual module 的网络层数都存在水分（GoogLeNet 官方宣称 22 层，ResNet 官方宣称 152 层），但是总体上的趋势还是很明显的，那就是网络结构向着复杂的方向演变，层数也向着变深的方向演变。
- 对于模型内存来说，除了 GoogLeNet（GoogLeNet 一般也是几个模型 ensemble 一起用），其他模型的体量都比较大，在前向计算时所花费的存储还是很大的。
- 模型参数是比较有意思的属性。看上去 VGGNet 的参数远超其他模型，实际上 VGGNet 的参数主要集中在全连接层上；而 GoogLeNet 和 ResNet 在全连接层上的参数并不多，所以参数总量看上去不大，而且它们的模型层数实际上比前面两个深很多，从这个角度看，模型的参数密度实际上是在减少的。

- 精度这里就不细说了，新模型的出现往往是由于精度的提升。

经过上面的分析，相信读者也了解了 CNN 网络发展的趋势，那就是从“Shallow and Wide”的模型结构转型成“Deep but Thin”的模型。模型的复杂程度不断增加，模型的拟合能力不断增强，但参数总量控制得很好，并不随之增长。152 层的 ResNet 和 5 层 conv+3 层 fc 的 AlexNet 模型参数数量相近，就说明了很多问题。

## 7.2 网络结构的演化

7.1 节提到了 CNN 发展过程中的几个经典模型，那么这些模型究竟是如何演化过来的呢？VGG 的“模型哲学”、Inception module 的思想、ResNet 对模型结构的颠覆是如何影响研究人员对模型结构的“三观”的呢？

### 7.2.1 VGG：模型哲学

在介绍 VGG 模型的论文中，作者介绍了 VGG 模型的几个特点，下面来详细介绍。

首先是卷积核变小。实际上，在 VGG 之前已经有一些模型开始尝试小卷积核了，VGG 模型只是成功案例中的一个。那么小卷积核有什么好处呢？文章中主要提出了两个好处。

1. 参数数量变少。过去一个  $7 \times 7$  的卷积核需要 49 个参数，而现在 3 个  $3 \times 3$  的卷积核有 27 个参数，看上去效果相近，但参数数量却降低了不少。
2. 非线性层的增加。过去  $7 \times 7$  的卷积层只有 1 层非线性层与其相配，现在有 3 个  $3 \times 3$  的卷积层配有 3 个非线性层。非线性层虽然不会增加模型的参数，但会增加模型的复杂度，这样模型的表现力反而有了提高。

文章还提出了 VGG 的模型收敛速度比之前的 AlexNet 要快些，这实际上和本层参数的数量相关。6.2 节曾分析过 CNN 模型参数的方差，假设对于某一层网络，这层的输入维度为  $N_l$ ，输出维度为  $N_{l+1}$ ，那么该层网络中每个参数的方差应该控制在  $\frac{2}{N_l + N_{l+1}}$  左右，这相当于给这一层的网络添加了一个无形的正则项。如果输入输出层的维度比较大，那么参数的理想方差就需要限定得更小，所以参数可以取值的范围就比较小，那个无形的正则项威力就会更强，优化起来就更费劲；如果输入输出维度比较小，那么每个参数的理想方差就会相对大一些，可以取值的范围就比较大，无形的正则项作用将减弱，优化起来也相对容易些。从这个角度看，减小每一层网络的参数数量对于优化来说是有意义的。



其次就是卷积层参数的数量的变化规律。在“VGG 哲学”中，卷积层的操作不应该改变输入数据的维度，这里的维度主要指 feature map 的长和宽。对于  $3 \times 3$  的卷积核，VGG 网络的卷积层都会配一个大小为 1 的 padding。同时 stride 被设为 1。这样经过卷积层变换，长宽没有发生变化。这和之前的卷积层设计思路有些不同。除此之外，每一次 Pooling 操作后，feature map 的长宽各缩小一倍，channel 层的数量会增加一倍。这样的设计对于不同维度的 feature map 来说适配起来都比较容易。对于一些通过卷积减小维度的模型来说，对于不同的输入尺度，卷积 Pooling 后的输出维度各不一样，所以模型不容易适配更多的场景；而现在只有 Pooling 层改变长宽维度，整体模型的维度计算就方便了许多。于是在论文中提到，对于维度为 256 和 384 不等的输入，模型不需要根据不同的输入维度设计不同的网络结构，使用同样的结构或者直接加深网络深度就可以适配。

此外，模型也提到了  $1 \times 1$  的卷积核。这种卷积核也不会改变 feature map 的长宽，只会在 channel 层次做聚合，这样又可以进一步增加模型的非线性层，增加模型复杂性的同时减少后续模型层的参数数量。

以上就是 VGGNet 在架构上做出的改变，这些改变也被后面的一些模型所接纳。

### 7.2.2 GoogLeNet: 丰富模型层的内部结构

GoogLeNet 模型的一大亮点是模型层的内部结构，这个模型最核心的地方就是它的 Inception Module。在此之前还有一个研究模型层内部结构的文章，叫做 *Network In Network*<sup>[5]</sup>，其中的道理也比较相似。

Network in Network 和 Inception Module 这类结构非常看中模型在局部区域的拟合能力。曾经有很多模型在结构上采用“一字长蛇阵”的方法，对于某一个特定尺度的数据，模型只采用一个特定尺度的卷积核进行处理。一张图像通常具有总体特征和细节特征这两类特征，小卷积核能够更好地捕捉一些细节特征，那么随着深层网络的小卷积不断地运算下去，总体特征也会慢慢地将一些总结提炼出来。

可是这里还存在一个问题，在这样的网络中，网络结构的前段一般只有细节特征，后段才慢慢有一些总体特征，这两种特征交替出现，并不重合。有时这两方面的特征重合在一起，同时发挥作用才会有效，那么采用单一维度的卷积核恐怕不太容易解决这样的问题。

上面提到的两种模型认为，采用一种尺度处理可能不太够，于是它们开始考虑把模型加厚，在每一次尺度的 feature map 上都尽可能地多做不同 scale 的分析，把想得到

的不同来源的信息尽可能地得到，这样的特征应该更有价值。Inception module 的一种结构形式如图 7-1 所示。

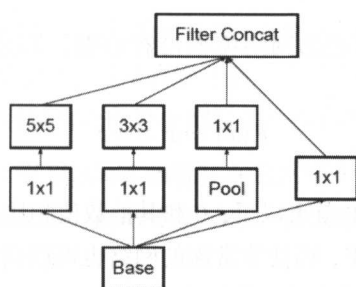


图 7-1 Inception module 的一种形式

以上就是它们带来的启发，模型除了要从深度下工夫，还要从宽度努力。

### 7.2.3 ResNet: 从乘法模型到加法模型

ResNet 的核心思想是把曾经 CNN 模型中的乘法关系转变成加法关系，让模型有了点“Additive”的味道。为了解释这个转变，论文中举了一个例子。

假设已经有了一个较浅模型，此时的目标是训练一个更深的模型来超过这个模型。理论上，如果能够找到一个优秀的优化算法和足够多的数据，那么这个更深的模型应该比那个较浅的模型具有更强的能力。如果抛开优化和可能的过拟合问题不管，这个理论还是可以成立的。就算较深的模型不能超越较浅的模型，它至少可以做到与较浅的模型同样的表达能力。如果把较深模型分成“和较浅模型相同的部分”与“和较浅模型相比多出来的部分”，那么只要保持第一部分的参数完全相同，同时让第二部分“失效”，只原样传递数据而不做任何处理，较深模型就至少可以与较浅模型效果一样，不会变弱。这些“失效”的模型部分被称作“Identity Mapping”，它们的输入和输出完全一样。

那么对于现在的模型来说，如果遇到这样的问题，模型要如何学习这些“Identity Mapping”呢？过去的模型结构采用自上而下的方式构建，因此层与层之间的关系是乘法关系，下一层的输出是上一层输入和卷积相乘得到的。想要学习这样的“Identity Mapping”，还是有点困难的，因为模型期望学到的是某个具体数值，这需要优化过程的配合。

于是，ResNet 对上面的问题做了一些改变。既然要学习“Identity Mapping”，那么能不能把过去的乘法转变为加法？假设多出来的层的函数形式是  $F(x)$ ，那么乘法关系下的“Identity Mapping”的运算公式为：

$$F(x) = \sum wx = x$$

它的参数优化方式和常规的网络模型没有差别，但是对于加法模型来说就简单多了。

$$F(x) = x + wx$$

这时只要将参数学习成 0 就可以了，0 和其他数值相比具有很大的优势，这样训练难度就大大降低了。这样一来，即使非常深的网络也可以训练，这也验证了将乘法关系改为加法关系后对模型训练带来的显著提升。

在 ResNet 之前，也有其他网络已经提出了类似的思想，比如 *Highway-Network*<sup>[6]</sup>。*Highway-Network* 同样具有加法的特点，但是它并不是一个纯粹的加法，它的网络结构的思想和 LSTM 有些相似。

## 7.2.4 全连接层的没落

从上面三个模型结构的演化可以看出，全连接层的地位在不断下降，模型中占据的比例在不断减少。全连接层和卷积层相比确实存在着一些劣势。

1. 参数数量多：VGGNet 中很多参数来自全连接，而全连接只占据前向运算的一部分，因此它的参数数量和运算量不成比例，看上去是个容易过拟合的地方。而且为了防止过拟合，全连接层一般还会配备一个 Dropout 层，这又为网络增加了运算负担。
2. 打破图像原本的维度：使用卷积层的网络可以很好地保持原始图像的空间位置，而到了全连接层，这些空间位置将不复存在。这样在一些多尺度训练的场景下，全连接层参数所表达的内容可能和不同尺度的特征错配，因此大家一般只在最后一层使用全连接层（或者使用 Average Pooling）。

关于全连接层和卷积层使用的讨论还在继续，虽然全连接层在图像领域的地位不及卷积层，但在其他领域还能发挥很重要的作用，所以对全连接层的研究仍需继续。

以上就是几个模型闪光点的回顾，如果读者想进一步研究这些模型及模型结构中的精妙之处，多做实验多分析数据才是王道。

## 7.3 Batch Normalization

本节将介绍当今深层网络的明星结构——**Batch Normalization Layer**<sup>[7]</sup>（简称 BN）。BN 层是一个口碑非常好的模型层，几乎存在于每一个当今知名的网络结构中。在介绍这个算法的原理之前，先回顾前面章节介绍过的一些内容和目前遇到的问题。

- 第 4 章和第 6 章中曾介绍 Sigmoid 函数在深层网络中的不足，在反向传导的过程中会有梯度消失的情况发生。如果梯度过小，模型就无法得到充分的优化，也无法像梯度爆炸那样进行补救。
- 第 6 章曾介绍参数初始化的重要性。由于深层网络通常采用相对固定的步长做参数更新，每层参数的数值大小和更新量差异最好不要太大，不然优化起来会产生很多问题。虽然现在有非常优秀的参数更新方法——如第 6 章介绍的 Xavier 和 MSRA，但是实战中一切并没有想象得那么美好。由于两个初始化方法都存在着一定的假设，当模型层数不多时，假设和真实情况的差距不大，模型优化不会有异常；当模型层数不断加深，假设场景和真实情况的差距越来越大，随着训练优化的不断进行，这些美好的假设就有可能崩塌，模型通常会陷入无法优化的境地。

Batch Normalization 的出现，很好地解决了上面提到的一些问题。

### 7.3.1 Normalization

相信读者也有所了解，只要对输入数据做简单地预处理就可以让模型优化效果有很大提升。例如，本书反复提到的 MNIST 数据集问题，如果用 Caffe 中自带的 example 进行训练，用不了多久就可以训练出一个 0.99 精度的模型。模型配置文件的 Data Layer 中，有这样一行配置：

```
transform_param {
    scale: 0.00390625
}
```

由于 MNIST 数据集中的像素值范围是 0 ~ 255，为了模型更好计算，数据的范围要压缩到 0 ~ 1。但如果把这三行注释掉呢？实验发现这个模型的准确率立刻掉到 10%，模型又和随机猜的效果等同了（上个精度为 0.1 左右的模型发生在 6.1 节，那个实验中参数的初始化被放大到了一个大的范围）。这说明输入数据的 scale 也是一个很重要的属性，深度模型虽然强大，但是如果不能把它调整好，给它足够优秀的数据，就不能发挥它的屠龙之术。

基于这个现象，有人想出了一个办法：既然这种 `rescale` 的方法这么灵，可不可以 在模型的每一层结束后将输出值都 `rescale` 一下？这样数值问题有可能就解决了。想要 达到这样的效果，就需要在每个卷积层的后面做一个白化操作：让所有的数据减去它 们的均值，再除以它们的标准差，这样数据又回到了原来非常理想的状态。

如果继续刚才的实验，去掉 `rescale` 输入数据的配置，然后给每一个核心层的输出 加上一个使数据白化的网络层（在 Caffe 中它叫做 `BatchNorm`），训练的精度达到 0.993， 模型收敛了不说，精度还比之前高出一些来，看来这个方法还挺管用。不过，这个实验 在小型数据集小型网络中做还可以，到更大规模的问题上就会失灵。

这种做法可能会遇到一个问题：它会破坏原有计算结果的分布，强行转变过来虽 然好训练但是算法可能就不对了。

于是在 BN 的作者看来，强行破坏原有计算结果的分布可能会造成不好的影响。于 是作者为 BN 加入了一个线性变换层，用来修补“白化”数据结果后造成的影响。从 理论上推导可以看出，加入的这个线性变换层可以让数据恢复到原来的样子。这样 BN 就变得更强大：需要发挥作用时可以纠正数据的分布，不需要时理想状态下可以化作 “Identity Mapping”。关于 BN 层的前向后向计算公式，论文中已经讲得非常详细，这 里就不再介绍了。感兴趣的读者可以阅读论文了解。

### 7.3.2 使用 BN 层的实验

下面通过实验来看一看 BN 的效果。首先来看看 BN 的配置该如何编写。论文已经 通过大量实验得出结论：Batch Norm 层要紧接在卷积层和全连接层的后面才能发挥最 大作用，这样就明确了 BN 的使用位置。Caffe 中 Batch Norm 层的功能并不完整，它只 包含一个“白化”功能，想要获得完整的功能，需要在 BN 层后面再接一个 Scale 层，也 就是论文中提到的线性变换层。Scale 层后面再接非线性层，比方说 ReLU 层。

这次实验的数据集是 CIFAR10，这是一个自然场景图片分类的数据集，共有训练 数据 50000 张，测试数据 10000 张，10 个类别。这里采用 ResNet-20 的模型解决这个图 片分类问题。ResNet-20 就是 20 层的残差网络。它的网络结构如图 7-2 所示。

可以看出，模型的每一层卷积操作之后都会配备一个 BN 层和 Scale 层。经过 64K 轮的训练，最终得到了 0.9 的 Test 集合精度。在没有做任何数据增强的情况下得到这样 的精度，也算是个十分不错的成绩了。如果没有 BN 层，直接训练 20 层深度的网络会 有些困难，而有了 BN 层，ResNet 也就能发挥出它的实力。

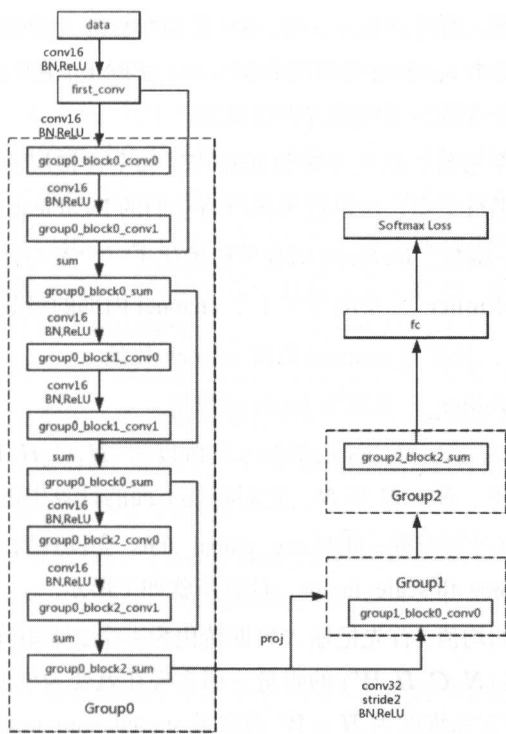


图 7-2 ResNet-20 的结构 ( 省略了中间重复的部分 ), 其中的 conv16 表示卷积核为  $3 \times 3$ , padding 为 1, 输出维度为 16 的卷积操作, proj 表示 ResNet 中的跨组投影操作

7.3.3 BN 的实现

7.3.2 节读者对 BN 算法的原理已经有了大概的了解, 本节将深入 Caffe 代码的内部, 看看 BN 层的实现细节。

总体来说, BN 层有一个输入, 一个输出, 它的输入输出可以保持一致, 很像非线性层。在这一层共有 3 个参数。前两个参数是大家熟悉的均值和方差, BN 需要在训练时记录下训练数据的均值方差统计信息, 然后将这些统计信息应用到预测数据上, 这样才能起到“白化”操作的一致性。由于 BN 的标准化是基于 channel 的, 所以参数的维度与输入数据有多少 channel 相关。第三个参数叫做 `scale_factor`, 它并不是论文中的主角, 但本节后面会提到它。

第 5 章曾介绍过一个 Caffe Layer 的核心方法: 初始化 ( `SetUp`, `Reshape` )、前向计算和后向计算。下面就按照这个顺序介绍 BN 层的前两方面的内容。

首先是 `Setup` 函数, 这里有两个值得一提的地方。一个是设置 `use_global_stats`, 它控制 BN 层的功能, 当它设为 `false` 时, BN 层的参数进入写状态, 每个 channel 的均值

方差统计信息将被更新，当它设为 `true` 时，BN 层的参数进入读状态，“白化”操作将根据参数进行，而不是每个 batch 自身的均值和方差。实际上，BN 层会根据所在的 phase 自动切换是否使用这个变量，无须读者自己设置。

其次是本层的三个参数，这三个参数同卷积层和全连接层的参数不一样，它们不需要通过梯度下降法进行更新，所以这里强行将它们的 `local learning rate` 设置成了 0。

接下来是 `Reshape` 函数。`Reshape` 函数里初始化了三个参与计算的变量。

1. `spatial_sum_multiplier_`：长度等于 1 个 channel 内部的参数数量。
2. `num_by_chans_`：长度为 channel 数量  $\times$  batch 数量。
3. `batch_sum_multiplier_`：长度为 batch 的数量。

我们举个例子，如果有一个 BN 层的输入的维度为  $(N, C, H, W)$ ，那么这三个变量的维度分别为： $H \times W$ 、 $N \times C$  和  $N$ 。它们将在下面的计算中发挥作用。

下面就是 Forward 计算流程，此时 `use_global_stats_` 会设置为 `false`，BN 层会修改当前存储的 `mean`、`variance` 和 `scale_factor`。计算步骤如下所示。

1. 计算当前 batch 的均值。首先把输入数据转化为一个 2 维矩阵，如果原本的输入数据是一个经典的  $(N, C, H, W)$  的张量，那么现在就变为维度为  $(N \times C, H \times W)$  的矩阵。接着把它和维度为  $H \times W$  的向量 `spatial_sum_multiplier_` 相乘，就得到了中间结果 `num_by_chans_`，它的维度是  $N \times C$ ，正好吻合，换句话说，这里把每个 channel 的数据加起来了。最后给这个乘积加一个系数  $\frac{1}{N \cdot H \cdot W}$ ，其中的分母是 `mean` 中相加的数字的数量。
2. 把 batch 中相同 channel 不同 num 的数据再做一次加和，这次就需要使用维度为  $N$  的向量 `batch_sum_multiplier_`，相乘后得到向量 `mean_`，这一步的归一化系数已经在上一步被除掉，因此这里不用再计算了。
3. 完成了求 `mean_` 的过程，现在要把 `mean_` 从这批数据中剪掉。这一次同样需要两步，先让 `mean_` 和 `batch_sum_multiplier_` 相乘，把结果存储到 `num_by_chans_` 中，然后再用 `num_by_chans_` 和 `spatial_sum_multiplier_` 相乘。这次由于相乘的顺序不同，一个  $C$  维的向量被扩展成了维度为  $(N \times C, H \times W)$  的矩阵，这个矩阵和输入数据维度相同，于是原始数据和 `mean_` 完成了相减。
4. 下面就要计算方差了。计算方差的过程比较巧妙。读者应该知道经典的方差公式为：
$$\text{var} = \frac{1}{N} \sum_i (x_i - \mu)^2$$
。在上一步中输入数据已经减去了均值，那么这里只要把这些数据按元素求平方，然后聚合求平均就可以得到方差。于是除了求平方这一步，其他的步骤和前面 1~3 步求解均值的过程没有区别。
5. 这里就要进行全局的均值和方差的更新。和其他做数值更新的问题一样，这里同样要考虑一个数值稳定性的问题，如果每一批输入数据的均值和方差的数值不够



稳定，时而大时而小，那么为了数值的稳定，这里采用多次出现在本书中的方法——滑动平均（Moving Average）。假设新的更新量为  $x$ ，历史累积的更新量为  $s$ ，那么有  $s_{t+1} = \gamma \cdot s_t + x$ 。其中的  $\gamma$  表示滑动平均中的打折率。可以看出，经过滑动平均，历史累积的数量会不断增加，数值很有可能超过任何一次计算得到的数字，那么将这样的累积量用在测试数据上就会出问题：数据将会按照超过平均值数倍的规模进行“白化”。为了解决这个问题，除了滑动平均均值和方差，参与滑动平均计算的数量也需要“滑动平均”计算。于是在每一轮计算中，`scale_factor` 也要进行这样的计算： $\text{scalefactor}_{t+1} = \gamma \cdot \text{scalefactor}_t + 1$ 。当 `use_global_stats` 设置为 `true` 时，只要每一个统计量都除以 `scale_factor`，数值就会保持稳定。

6. 整个前向计算就只剩下一个和概率论有关的问题。基于样本计算随机变量的方差时，需要计算“无偏估计”的方差，这要在之前求出的方差基础上乘以一个系数，这样求出的才是基于样本的无偏估计的方差。

以上是训练时的计算流程。当 `use_global_stats` 设置为 `true`，进入预测过程时，它的流程又会变成什么样呢？了解了上面训练的流程，它的过程就简单多了，只要减去统计的均值更新量，除以方差统计量即可。

到这里，BN 层的计算流程已经介绍完毕。图 7-3 总结了上面介绍的流程。

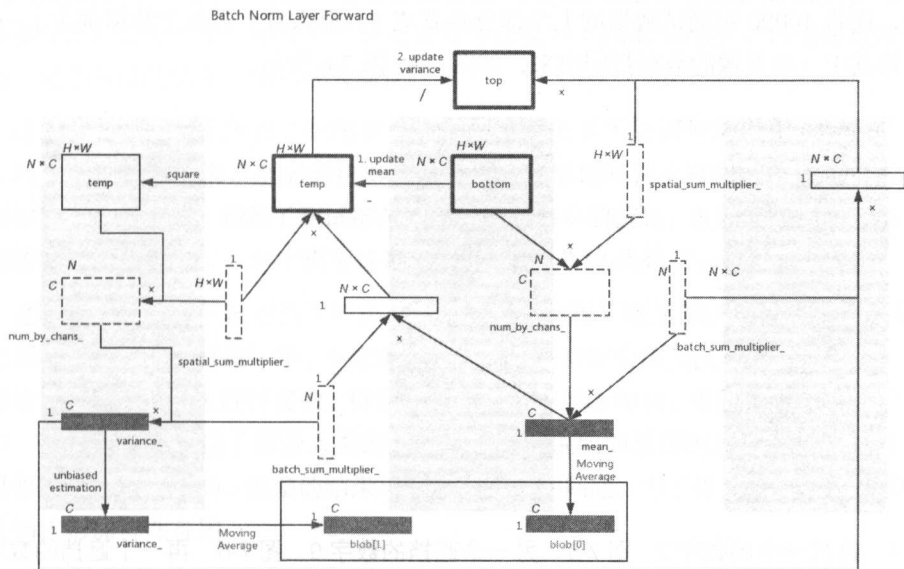


图 7-3 BN 层计算流程。其中粗线框表示从 bottom 到 top 的更新过程中涉及的变量，实心框表示参数保存涉及的变量，虚线框表示局部变量（有重复）

以上就是对 BN 的介绍，BN 从网络内部入手解决了数值不稳定的问题，但同时又给自己留好了“后路”，确保不会过度修改模型中的数值。这些优势就是它能够被广泛



应用的原因。当然，如果网络本身没有数值问题还是不要加入 BN 层，因为 BN 层会带来额外的计算量。

## 7.4 对 Dropout 的思考

说起深层模型和卷积神经网络，就不得不提到在模型中有着很重要作用的 **Dropout** 层<sup>[8]</sup>。Dropout 层最经典的用法是放在全连接层的前面在。训练阶段，全连接层的输入首先经过 Dropout 层，这样一部分输入数据会被 Dropout 层随机丢弃而置 0，不参与本次计算。每一轮迭代中 Dropout 层丢弃的数据都可能不同，所以每一次模型的表现也会不同，被更新的参数都不一样，而且这些保存下来的数据的梯度在反向计算时也得到了相应的增强。在预测过程中不再开启 Dropout 层，而是让每一个数据都发挥作用，同时也不再给任何一个数据做权重增强。这时，经过 Dropout 层训练的模型会具备更好的效果。一般来说，Dropout 拥有降低数据过拟合风险的性质，同时还拥有模型融合的效果，似乎亲身实践了“Less is more”这句格言。为了验证 Dropout 产生的效果，本节将做一个比较激进的实验——半字识别。这次实验的主角还是熟悉的 MNIST 数据集，为了让这个实验变得足够有趣，实验数据需要做一些变化。实验中保持 60000 张训练数据不变，而将 10000 张测试数据的上半部分重置成 0。那么看上去每个数据都少了一半。这里将其中一些图像的数据打印出来，如图 7-4~ 图 7-6 所示。

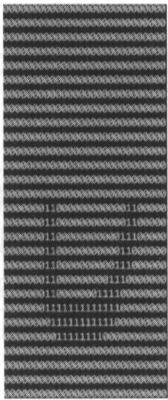
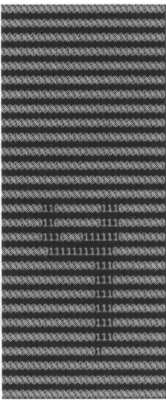
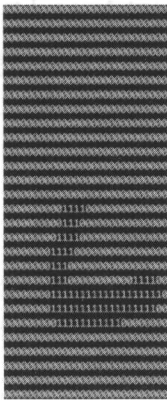


图 7-4 遮挡一半的数字 2 图 7-5 另一个遮挡的数字 9 图 7-6 再一个遮挡的数字 0

对人类来说，猜出这些图像的数字虽然并不算特别难，但是和识别正常的数据相比，也算是一个有难度的问题了，而且上面给出的图像实际上只是比较简单的一些图像，还有很多难到几乎无法给出正确答案的图像，例如上半部分遮挡的“1”和“7”。这种人类都觉得困难的问题，交给计算机恐怕也是很有挑战的。这个问题实际上也算

是识别问题中遇到的一个十分经典的问题：物体遮挡（occlusion）。如果待识别的物体被遮挡了一部分，模型还能认出它来吗？对人来说，只要没有把所有有用的信息都遮挡住，就可以通过局部信息识别出一个整体的物体。能做到这一点，说明人类具有利用部分信息进行分析推断的能力。如果希望计算机拥有人的智能，那么它最好也可以拥有这样的能力。

我们先利用前面使用过的 LeNet 模型看看它在这个问题上的表现。LeNet 模型的结构如图 7-7 所示。

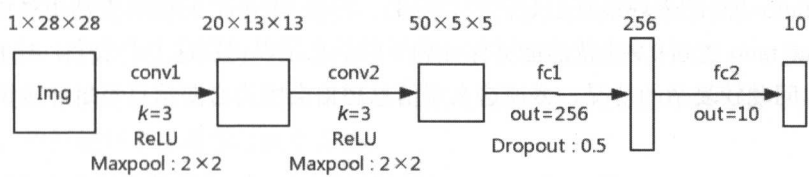


图 7-7 识别遮挡数字的模型结构

利用上面提到的训练数据进行训练，最终得到的测试集精度为 0.4358。识别率不到一半。实际上，如果观察在训练过程中的日志信息，可以发现训练过程中某些轮次的测试集精度比这个数高一些，但确实会出现越训练效果越差的情况。这说明模型对训练数据存在过拟合现象。虽然早于预定轮数停止模型（Early Stop）可以缓解过拟合的问题，但是取得的精度还是不令人满意。

这个问题的根本原因在于训练集和测试集实际上并不是同样的数据分布和信息容量。因为在训练时给出了全部的数据信息，所以在识别时每个有用的信息都会被当作识别的特征加以训练；而到了测试部分，数据只有一半的信息，曾经学习到的一些很有把握的特征突然消失了，对于模型这样的“耿直 boy”必然是“一脸茫然”。

这时聪明的读者一定想到了解决方法，有舍才有得！既然测试数据只有一半信息，那把训练数据也变成只有一半，大家的信息一致，模型就会更关注这部分的信息。基于这样的设想，我们再次进行实验，得到的测试集精度为 0.9044。果然比之前的结果高了不少，甚至有可能高过了部分人类的识别水平。这说明如果训练过程的数据特性和测试过程的数据特性不同，模型的结果可能会有很大的问题，为了提高效果，有时删除特性反倒比增加特性有用。

当然，如果能发现问题并想出自断一半数据的方案固然好，但如果没有发现这样的情况呢？这时不妨用 Dropout 的思想来解决，由于每次训练时数据只提供一部分信息，如果模型也只使用部分信息进行推断预测，那么就更适应测试数据的场景。下面的实验中，模型的 ip1 层的后面将加上 Dropout Layer，并测试 dropout\_ratio 从 0 到 0.9 的效果，最终结果如图 7-8 所示。

dropout 比率	0.9	0.8	0.7	0.6	0.5	0.4	0.3	0.2	0.1	0
精确率	0.66	0.62	0.58	0.62	0.59	0.53	0.52	0.51	0.45	0.44

图 7-8 Dropout ratio 和精确率之间的关系

从图 7-8 中的结果看，dropout 比率为 0，守着所有特征不放的模型精度最差，为 0.44；而 dropout 比率为 0.9 的模型表现最好，精确率为 0.66，实验显示 dropout 比率越高，精度越高。这么看来，“Less is More” 这句格言似乎有点道理。不过在这个例子中，dropout\_ratio=0.9 的表现最好也是比较特殊的，因为 MNIST 的输出类别相对较少，即使 dropout\_ratio 达到 0.9 也依然能够保证剩下的信息足以识别这十个数字，对于一些类别较多，问题较复杂的情况，丢掉这么多信息恐怕会因为必要信息不足导致识别精度下降。

介绍 Dropout 的论文提到了 Dropout 的两种好处之一：减轻模型过拟合的情况。相信读者可以从上面的实验中体会到其中一二。但是，解决问题正确的方式是让训练集合和测试集合的数据保持一致性，这比加入 Dropout 层要重要得多。

## 7.5 从迁移学习的角度观察网络功能

前面我们介绍了许多有关网络结构的内容，主要是站在模型的外面，把模型当成一个黑盒进行实验分析，我们很难说清楚模型究竟做了些什么事情。其实可解释一直是机器学习模型的一个软肋，简单的模型可解释性强，但是能力相对差一些；复杂的模型能力强，但是解释性又差得一塌糊涂。似乎模型能力和解释性是一对矛盾体，而这对矛盾体到了深层模型这里变得更严重，我们获得了更强的模型，也获得了更让人一头雾水的计算方法——为什么这样乘这样加就能得到正确结果，你的理由呢？

深层模型除了加入了不可解释的大军，还提供了另一串让人发蒙的问题：如何界定层与层之间的功能？第一层是干什么的，第二层是干什么的？它们的功能有没有重复的部分？这些问题也成了一些浅层模型拥护者的吐槽点。实际上科研人员也一直在这条道路上不断前进，本节就来看看其中的一个研究——*How transferable are features in deep neural networks?*<sup>[9]</sup>。

一些讲述模型的学术论文经常会提到一句话：“我们的模型第一层的参数和 Gabor Filter 的参数十分相似”（当然，这句话一般都是用英文说的），这似乎是在表示作者的模型拥有经典图像算法中具有的优秀性质。由于 Gabor Filter 这样的滤波器属于比较通用的滤波器，因此上面这句话是想告诉大家：我的模型在一开始也要完成一些常规模式的计算过程，这一点和那些经典算法是一样的。于是论文的读者就会明白，这个模型

的第一层会完成通用的图像分析工作。当然，这里也在向大家暗示，深度学习的模型也是可以解释的。

以此类推，到了模型的深处，那些较深的模型层会完成一些更专业的工作，也就是和训练任务紧密相关的工作。这种解释非常容易让人理解，因为这样的思路 and 人们处理事情的方式很像。遇到一件事情，人也喜欢先用成熟通用的模式解决，随着问题的不断深入，再加入一些与问题紧密相关的方法，这样才能更好地完成这件事情。

既然我们已经接受了这种处理问题的模型，那么下面就要解决另一个更细致的问题——哪些层次是用来解决通用问题的，哪些层次是用来解决具体问题的？除此之外，通用问题和具体问题的界限在哪里？解释清楚这些问题对我们深入理解深层模型的内部机制是有帮助的。那么，这里应该用什么方式展示模型层在这方面的能力呢？“很自然”地，我们想到了迁移学习这个方法。

**迁移学习**的目标是以模型作为媒介，将在一个领域中学到的知识应用到另一个领域上，如果两个领域之间的共性比较大，那么迁移学习的效果就会很好。同样，如果一个网络层比较通用，那么迁移到一个新的问题上也会比较好用。

这里假设有两个数据集  $A$  和  $B$ ，其中  $A$  和  $B$  的内容比较接近，如果采用迁移学习的方式进行学习，无论是从  $A$  到  $B$  还是从  $B$  到  $A$ ，理论上，从一边学到的模型都会在那边发挥较好的功能。于是读者也可以想象，如果设计一个同时适用于数据集  $A$  和  $B$  的模型，首先让模型在数据集  $A$  上学习，然后将模型在数据集  $B$  上继续微调参数 (finetune)，那么模型在数据集  $B$  上一定会有比较好的效果。

除了微调参数，迁移学习还可以有两个极端的选择。一种是完全抛弃原有的参数，重新训练，一种则是完全固定参数，不进行训练。显然，微调参数处于这两种情况之间。可以想象，面对上面的选择，不同的模型会有不同的表现。对于迁移能力或者泛化能力很强的模型层来说，即使模型的参数完全固定，在不同的问题上它的表现也不会差，像前面提到的那些像 Gabor Filter 的模型层就具备这样的性质；而对于迁移能力不强的模型层来说，固定它将无法完成迁移的工作。这也说明模型在这一层的通用性不是很强。在遇到新问题时，这一层的参数需要被重新训练。通过上面的分析，我们可以认为利用这样的实验可以帮助判定模型层的通用性。

下面就来看看这篇论文的实验。相信这篇论文中的实验会对我们有很大的启发。实验使用的数据集是 ImageNet 数据集，所做的任务是图片分类任务。实验将整个数据集随机分成两个子数据集，并以这两个数据集创建两个识别任务—— $A$  和  $B$ 。由于两个任务的数据是从同一个数据集中分离出来的，因此两个任务的相关性非常大，从一个任务学到的模型很有可能可以在另一个模型中发挥作用。

基于这个猜想，下面我们就要完成两组实验。

首先是同任务的再训练：在任务 *B* 上训练后，将模型做一些处理，继续在任务 *B* 上做训练。处理的方式有以下两种。

- 1. 固定其中一部分参数，并将其他模型层的参数做随机初始化，然后继续在任务 *B* 上训练。
- 2. 保留其中一部分的参数但不固定，并将其他模型层的参数做随机初始化，然后继续在任务 *B* 上训练。两个实验的模型图如图 7-9 所示。

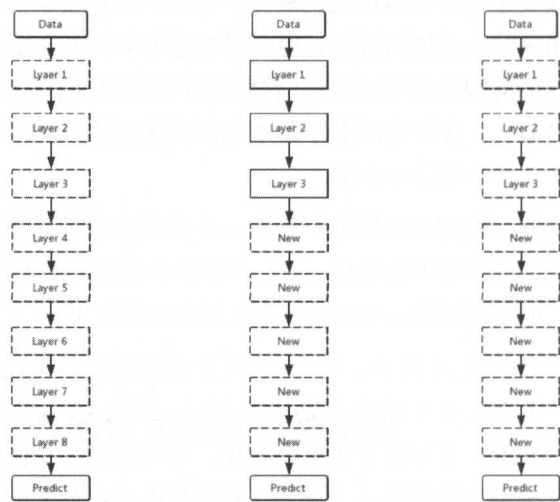


图 7-9 迁移学习的第一组实验方案

图 7-9 展示了三个模型，最左边的模型是由原始数据训练得到的；中间的模型将前三层参数固定，其余层参数初始化重新训练；最右边的模型保留前三层参数作为初始化参数，其余层参数重新初始化，所有参数重新训练。实线框表示参数固定的层（数据层和目标函数层也可以当作参数固定），虚线层表示参数可变的层。这两个实验的最大区别在于初始化方式和参数固定的方式。两个实验都要将已训练的模型作为初始化的值，这个数值和常规的初始化方法可能不同；另外，其中一个实验要将部分参数固定，相当于测试这部分参数的通用性。

为了还原论文中的实验，如表 7-2 所示列出实验中的数据（数字可能有微小的差异）。

表 7-2 迁移学习第一组实验结果

固定层数	原结果	1 层	前 2 层	前 3 层	前 4 层	前 5 层	前 6 层	前 7 层
实验 1	0.635	0.635	0.632	0.626	0.618	0.595	0.625	0.633
实验 2	0.635	0.618	0.623	0.636	0.632	0.635	0.634	0.632

从实验结果看，实验 2 的效果更令人满意。由于并没有固定任何一层模型，即使从当前模型出发进行训练（虽然会有小的波动），最终模型还是会得到一个相对不错的结果。然而实验 1 的结果就又变得有些奇怪了。固定前三层训练的结果整体表现不错，到了第四第五层，模型呈现了明显的下降趋势，到了第 6 层、第 7 层，模型又恢复了正常。这是不是说明中间几层的模型通用性不太好呢？

可以说，中间几层的通用性确实不好，不但如此，由于处于十分关键的位置，中间几层的特征变换还存在着很强的依赖性。读者可以想象，模型的前几层主要完成通用的特征变换，因此即使固定它们，效果也不会太差；到了最后几层，由于模型的特征已经基本构建完成，这时问题已经从一个复杂的高维非线性问题变成了相对低维的线性问题，这时的问题也只需要通用的线性分类方法即可完成；最困难的实际上是中间的步骤，这里是完成专用特征变换的关键位置。由于前面几层模型完成了通用的处理，接下来的任务要依照这些通用的结果进行计算，此时可选择的计算方式已经不多了，所以能与前面计算结果搭配的参数本来就不多。因此找到与之搭配的参数成了关键，这实际上比重新训练一个新模型还要困难。

这个现象被称为“**fragile co-adapted features**”，作为一个流水线，相邻层之间的参数存在着很高的耦合性，它们往往是通过协同学习得到的，而不是各自单独学到的。因此将它们拆开学习可能会得到不好的结果，而实验结果也揭示了这一点。

其次是做不同任务的再训练，即我们在任务 A 上训练后，将模型做一些处理，然后在任务 B 上做训练。处理的方式同样有两种。

- 1. 固定其中一部分参数，并将其他模型层的参数做随机初始化，然后继续在任务 B 上训练。
- 2. 保留其中一部分参数但不固定，并将其他的模型层的参数做随机初始化，然后继续在任务 B 上训练。

这两个实验的过程和上面两个实验相同，唯一不同的就是训练的参数。两个实验的数据如表 7-3 所示。

表 7-3 迁移学习第二组实验的结果

固定层数	原结果	1 层	前 2 层	前 3 层	前 4 层	前 5 层	前 6 层	前 7 层
实验 1	0.635	0.636	0.631	0.627	0.613	0.582	0.585	0.553
实验 2	0.635	0.638	0.642	0.647	0.644	0.644	0.645	0.646

这里的实验 1 展示了固定参数的迁移学习实验，可以看出，前面 3 层参数的固定并没有太多影响模型效果，可见这三层模型的通用性效果很好，但到了后面，模型的通用



性越来越差，同时层与层之间的耦合性也被打破，精确率也不断降低。由于任务  $A$  和  $B$  的数据不同，这个下降是可以理解的。

实验 2 展示了真正的迁移学习实验，可以看出模型的效果基本保持甚至有所提高。这是迁移学习中表现出的正常效果。同单独训练任务  $B$  相比，先训练任务  $A$  再训练任务  $B$  竟然获得了更好的效果，这说明训练任务  $A$  的数据确实起到了正向的作用。而且如果将这个实验的结果和上一组的结果做对比，我们还可以发现这个精确度的提升并不是因为模型经过了更多的训练，上一组实验的实验 2 也经过了两轮的实验，但是效果并不比这一轮的实验 2 好。

那么这个效果的优异来源于哪里呢？实际上当初数据  $A$  和数据  $B$  被随机划分开，两组数据的相似度很高，因此实验 2 相当于接触了更多的训练数据，因此表现也更为优异。但面对更多的数据，固定不同层数的参数会有不同的表现。

- 实验验证了模型前几层具有更好的通用性，但实际上面对更多的数据，它们的效果也得到了提升，说明在这些模型层中，专用性同样存在；
- 实验同样验证了模型后几层的效果，从第 4、5 层开始，固定参数带来的收益已经很小了。可以看出后面几层对数据的敏感性比较弱，它们的表现更依赖于前面模型层在抽取特征方面的表现。

虽然原论文中在后面还有其他实验，但本节只引用这个实验。从上面这两组实验中，我们可以看到一种分析模型层宏观特征的方法，这些内容足以帮助我们打开一扇分析模型方法的大门。实验揭示了模型层在通用性方面的表现，同时展示了线性模型结构中模型层之间的依赖关系。虽然它没有解释更多模型层深入的特性，但是相信在未来的道路上我们可以对模型有更多直观的了解，也能为深层模型摘掉玄学这个帽子。

## 7.6 ResNet 的深入分析

2015 年问世的 ResNet 完全颠覆了很多研究人员对深层模型的理解。大家曾认为模型的深度是存在极限的，就像人脑一样，虽然神经元存在层级关系，但层级的总数一定是有界的。而 ResNet 向大家展示的结构似乎是无界的：模型的深度可以达到上千层也不会出现训练上的问题。然而从模型的实际表现来看，更深的模型并不会带来等量的精度提升，即使深度增加了几倍，所带来的精度的提升也比较有限。这个现象和 ResNet 论文中的解释十分相近：更深的网络可能只学到了 Identity Mapping，而不是真正的进一步的特征变换。本节将介绍和 ResNet 相关相近的一些研究成果，进一步揭示 ResNet 网络和相关网络的特性。

7.6.1 DSN 解决梯度消失问题

其实深层模型梯度消失的问题由来已久，本书的很多章节也一直在讨论这个问题。由于网络结构采用线性堆叠的方式，后一层网络参数的梯度必须依靠前一层网络的梯度，这样的梯度依赖就会产生很多数值上的问题。为了解决这些问题，研究人员想出了很多解决办法，其中一种解决方法来自论文 *Deeply-Supervised Nets*<sup>[10]</sup>。

这篇文章解决梯度问题的思路十分清晰：既然分类模型的目标是要将不同种类的物体分开，那么到了最后一层，不同种类物体的特征必然具有可区分性（这里需要线性可分了）；往回推导，倒数第二层肯定也具有一定的可区分性，如果它不具有区分性，那么后面的变换是无法将它们区分开来的；同理，每一层特征都应该具有可区分性。

但是这些特征具有的可区分性程度究竟如何呢？有没有可能出现这样一种情况：某一层的可区分性比较弱，但是其他层的可区分性比较强，最终的分类效果同样令人满意，而区分性弱的那一层表现不佳是因为梯度传导的问题？同时在 7.5 节中读者已经可以看出，线性结构模型中相邻层会出现相互依赖的问题，这些依赖会不会限制某一层的能力？为了解决梯度传导和模型层耦合的问题，论文的作者提出了为每一层增加判别目标的方法。

以往我们看到的模型一般只有一到两个目标函数，而且这些目标函数都在网络的结尾处。而在 DSN 中，每一个卷积层/全连接层过后，模型都会为它们接一个旁路的全连接层，并用输出的结果和数据的真实结果做比较，计算预测结果和真实结果的误差，并将由此产生的梯度回传给前面的参数。网络结构如图 7-10 所示。

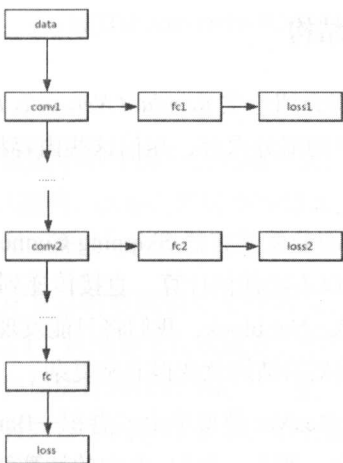


图 7-10 DSN 网络结构图



这样的结构很好地解决了前面提到的两个问题。首先，由于每一层都可以和目标函数“亲密接触”，因此即使模型很深，从后一层网络传来的梯度不够好，模型也可以通过这一旁路目标弥补梯度的不足；其次，同样由于每一层和目标函数近距离相连，每一层网络除了要和后面的网络配合完成分类任务，它也要具备独立完成分类的任务，因此模型层具有更强的独立性。

当然凡事也不能只看好的一面。虽然我们希望模型能够同时具备这两种能力：既能独当一面，又能打配合，但是这两种能力还是存在着一些冲突。为了平衡两方面的能力，作者也想出了一些比较直观的解决方法。如果一定要比较两个能力哪个更重要，相信读者也很清楚，当然是配合更重要，这也是深层模型超越浅层模型的地方。所以模型优化的目标一定是提升“合作能力”为主，提升“独立能力”为辅。下面两个方法正是这一思想的体现。

首先，对模型中间的旁路目标函数降权。每一个目标函数在计算时要加上一个函数权重，理想状况下，越是靠近真正的目标函数，权重应该越大，越远离真正的目标函数，权重应该越小。

其次，由于无法奢求中间的目标函数像真正的目标函数那样表现优异，因此模型需要容忍这些中间目标函数存在一些小偏差。为此模型为每一个中间目标函数设置了一个阈值，只有目标函数的损失超过了这个阈值，梯度才会被传导；如果没有超过，那么就认为这一层模型表现正常。

有了这两个方法，模型的训练目标就十分清晰了——二流的个人能力和一流的合作能力。实际上这个思想和 ResNet 是很相近的。

## 7.6.2 ResNet 网络的展开结构

有了前面的铺垫，本节将介绍论文 *Residual Networks Behave Like Ensembles of Relatively Shallow Networks*<sup>[11]</sup> 中的部分内容，相信这些内容将会使读者对 ResNet 有更加深刻的认识。

ResNet 网络中最精华的部分就是它的 Skipping Connection。这个结构使得前一层网络的特征信息/数值信息可以不经任何计算，直接传递到下一层，这在之前的网络中实属少见。如果只看某一个 ResNet block，我们将只能发现其中的加法特征，但是如果将多个 block 综合起来看，模型的结构就变得非常复杂。

如果用一个函数  $F$  表示 ResNet 模型中由“卷积—Batch Normalization—非线性计算 (ReLU)”组成的操作集合，那么一个 block 中的计算就可以由下面的公式表示：

$$X_{t+1} = X_t + F_t(X_t)$$

如果进一步推导多个 block 的运算，就有

$$\begin{aligned} X_{t+2} &= X_{t+1} + F_{t+1}(X_{t+1}) \\ &= X_t + F_t(X_t) + F_{t+1}(X_t + F_t(X_t)) \end{aligned}$$

可以看出，随着 block 不断堆叠，模型的加法特性会展现得更明显，很多特征信息将直接传导到模型的结尾处，这个特性和传统的线性模型结构完全不同。

这种结构带来了什么好处？实际上这个结构和 7.6.1 节介绍的 DSN 模型的思想非常相近，只不过形式不同。通过这样的结构设计，每一层网络都有了和最终目标函数“近距离接触”的机会，这样便于梯度的传递；同时最终的特征组合既包含了多个网络层的组合，也包含了一些网络层的单独行动，因此每个模型层的“个人能力”也得到了锻炼。

当然，两个模型还存在着一定的不同。在 DSN 中，增加的那些中间目标函数实际上在预测时可以去掉，而 ResNet 的结构相对复杂，在预测时已经无法将“个人能力”的部分剥离出来，因此在 ResNet 模型中，“个人能力”，或者说“小团体能力”还是会影响模型的整体发挥的。

另外一个重要的地方就是模型的加法特性。如果把函数拆开来看，就会得到：

$$\begin{aligned} X_{t+1} &= X_t + F(X_t) \\ &= X_t + \text{relu}(\text{BN}(\text{conv}(\text{relu}(\text{BN}(\text{conv}(X_t))))) \\ &\geq X_t \end{aligned}$$

也就是说，后一层的输出值不小于前一层的输出值，这在数值上并不是一件令人开心的事情。因此读者也可以猜测，随着模型层不断加深，计算得到的残差的数值应该是越来越小，而不是越来越大，否则不断累积的数字会造成数值膨胀甚至溢出。

如果多出来的一层不能让自己的数值幅度变得太大，那么多出来的一层有什么样的功效？这个问题就很有意思了。从模型上直观地看，每多出一个 ResNet block，模型将多出许多路径来，这让最终的加法计算项目多出了一倍，这种加法的形式和 Ensemble 的形式有点相近。论文的作者对这个问题有做了许多有趣的实验，感兴趣的读者可以详细阅读。作者最终发现了一个令人惊奇的结论，实际上随着 ResNet 的模型加深，模型在“大团队协作”方面的能力上涨的十分有限，支撑模型能力继续上涨的是“小团队

协作”能力。而小团队所构成的网络深度在 20 层左右，这个数字和 VGG 模型的深度很接近。

实际上“小团队”在很深的模型中也比较常见。对于 50 个 block 组成的网络，由 10 个 block+40 个 Skip Connection 组成的路径有多少呢？ $C_{50}^{10}$ 。这个数目相对而言是比较大的，所以每增加一个 block，“小团体”的数量会增加许多，其总体实力也会增长许多，而对于“大团体”的实力增长帮助很小。

这个发现和人类的组织结构存在着很多相近之处，因此也十分容易被人理解接受。现在军事上的人员组织也是从小团体开始划分，公司的项目组人数也不会太大，每一个新人加入团体也是直接加入到某一个小团体中，很难和大团体的所有人形成良好互动。当然，这个结果实际上也有点悲观。深层模型的强大之处在于各网络层的协作，在之前的很多模型中，我们看到了这种协作，甚至认可了其中一些协作的意义，但是协作也存在着某种上限，而且在很短的时间里科研人员已经碰到了这个上限。ResNet 网络似乎绕过了这个上限，为大家带来了更深的网络，但是无论从理论还是实践，ResNet 在发挥“深度学习”特点这件事上也略显乏力。不过这并不妨碍它成为一代经典模型，它以一种比 DSN 更优雅的方式解决了梯度消失的问题，这已经很值得敬佩了。

### 7.6.3 FractalNet

前面提到了 DSN 模型和 ResNet 模型在解决梯度消失时采用了同样的思想：让每一层参数都亲密接触目标函数，于是借着这个思想，一些新的模型也诞生出来。其中一个比较有趣的模型来自 *FractalNet: Ultra-Deep Neural Networks without Residuals*<sup>[12]</sup>，它提出了一个新的模型结构——分形网络。

有关分形的内容这里不去多说，它的模型也像 ResNet 一样通过一个一个 block 连接而成，但不同的是模型中没有 Skip Connection 这样的结构。每一个 block 结构可以用下面这个递归函数表示：

```
def simple(x):
    return relu(bn(conv(x)))

def fractal_net(x, n):
    if n == 0:
        return simple(x)
    else:
        return simple(x) + fractal_net(fractal_net(x, n-1), n-1)
```

这个递归函数可能不够直观,下面将画出  $n=4$  时的网络 block 结构,如图 7-11 所示。

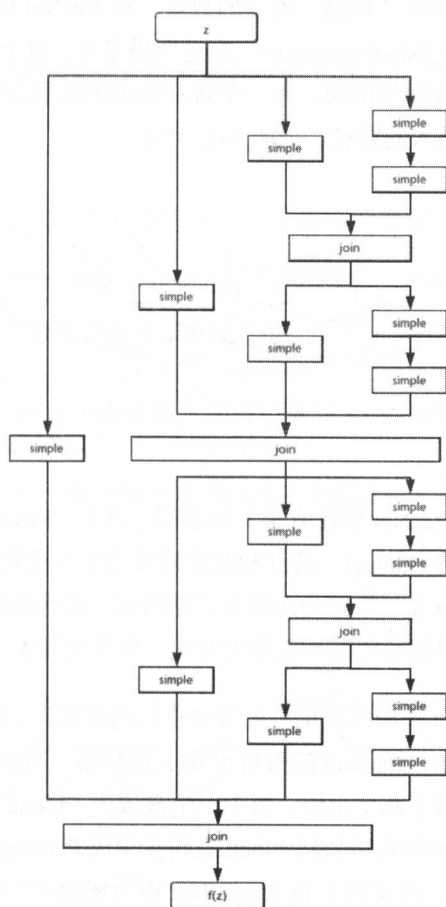


图 7-11 DSN 网络结构

从图中可以看出, FractalNet 吸收了 ResNet 模型的特点, 模型参数和目标函数贴近, 多条路径融合。但是它的模型结构比 ResNet 复杂一些, 同时, 它采用了 drop-path 的方法进行训练, 也是为了证明模型中的“小团体”实力不凡。将 ResNet 中的一些性质对应到 FractalNet 上, 就可以明白这个模型的特点, 这里就不再赘述了, 感兴趣的读者可以阅读论文。

#### 7.6.4 DenseNet

本节介绍的模型结构被称为“Densely Connected Networks”, 简称 DenseNet。模型出自论文 *Densely Connected Convolutional Networks*<sup>[13]</sup>。ResNet 的 Skip Connection 结

构相对比较简单，block 计算前的特征和 block 计算后的特征相加，形成了两条路径。DenseNet 最大的区别，就是“短路”连接的数量。在 ResNet 中，每个 block 中有两组卷积运算，其中包括 1 个 Skip Connection。在这个模型中，每个 block 包含  $N$  组卷积运算，组内的卷积层结果按顺序排列，每一个排在前面的卷积层的输出都会有一条直接连到它后面的卷积层的输入的路径，如图 7-12 所示。

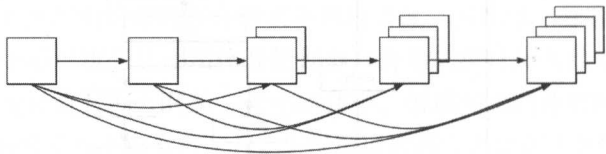


图 7-12 DenseNet block 计算结构图（箭头表示 conv-BN-ReLU 运算）

DenseNet 的结构和 ResNet 相比有很大的不同。首先，ResNet 中的短路连接方式是相加，相加过后，结果的 channel、高度和宽度均没有发生变化，但是数值的范围有可能发生变化。而 DenseNet 中，短路的连接方式是拼接，所有的特征数据按 channel 排列起来。使用这种结构，数值的范围不会发生变化，但维度发生了变化。

其次，由于 DenseNet 采用维度堆叠的方式进行特征组合，随着卷积层数不断加深，短路连接的个数不断增加，channel 的数目也在不断增加。当然模型不可能无限地增加，DenseNet 将模型分成了几个小组，每个小组被成为“block”，所有的短路只发生在 block 之内，而不会跨越 block。同时，block 之间会完成 Pooling 的操作。所以模型的结构借鉴了 ResNet 的形式，但实际上和 FractalNet 更为相近。

由于 DenseNet 增加了许多短路连接，网络变得更加稠密复杂。每一个网络层获得梯度的方式也变得更多，这带来了一个好处——DenseNet 的参数与同类模型相比偏少，同时它的识别效果又是同类型中最好的。这说明模型层之间的相互连接对模型学习的好处。

以上四个小节介绍了 4 种模型，读者可以看出其中的共性——它们都有让各模型层方便获得目标函数梯度的“短路径”。这种思维模式似乎成为近一两年模型设计的一个主流模式。当然，四个模型各自还是有自己的一些特点。了解了这个流行元素，读者更容易理解很多同类型的模型。当然更重要的是，让我们一同期待模型设计思想的下一次升华——新的流行元素是什么呢？

## 7.7 总结

本章主要从网络结构的角度分析了卷积神经网络的一些特点，让我们一起回顾一下。

- CNN 演化的过程。
- VGGNet 的思想、模型子结构的思想，加法模型的思想。
- ResNet 结构虽然奇妙，但还是存在自己的局限性。
- BN 在解决网络内部数值方面有很好的作用。
- Dropout 可以缓解模型过拟合的状况。
- 不同的模型层存在不同的通用性和专用性。
- ResNet 类的网络结构在解决梯度传导问题上的特点

## 7.8 参考文献

[1] Krizhevsky A, Sutskever I, Hinton G E. ImageNet classification with deep convolutional neural networks[C]// International Conference on Neural Information Processing Systems. Curran Associates Inc. 2012:1097-1105.

[2] Simonyan K, Zisserman A. Very Deep Convolutional Networks for Large-Scale Image Recognition[J]. Computer Science, 2015.

[3] Szegedy C, Vanhoucke V, Ioffe S, et al. Rethinking the Inception Architecture for Computer Vision[J]. Computer Science, 2015.

[4] He K, Zhang X, Ren S, et al. Deep Residual Learning for Image Recognition[J]. 2016:770-778.

[5] Lin M, Chen Q, Yan S. Network In Network[J]. Computer Science, 2013.

[6] Srivastava R K, Greff K, Schmidhuber J. Highway Networks[J]. Computer Science, 2015.

[7] Ioffe S, Szegedy C. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift[J]. Computer Science, 2015.

[8] Srivastava N, Hinton G, Krizhevsky A, et al. Dropout: a simple way to prevent neural networks from overfitting[J]. Journal of Machine Learning Research, 2014, 15(1):1929-1958.

- [9] Yosinski J, Clune J, Bengio Y, et al. How transferable are features in deep neural networks?[J]. Eprint Arxiv, 2014, 27:3320-3328.
- [10] Lee C Y, Xie S, Gallagher P, et al. Deeply-Supervised Nets[J]. Eprint Arxiv, 2014:562-570.
- [11] Veit A, Wilber M, Belongie S. Residual Networks Behave Like Ensembles of Relatively Shallow Networks[J]. 2016.
- [12] Larsson G, Maire M, Shakhnarovich G. FractalNet: Ultra-Deep Neural Networks without Residuals[J]. 2016.
- [13] Huang G, Liu Z, Weinberger K Q. Densely Connected Convolutional Networks[J]. 2016.

# 8

## 优化与训练

作为机器学习中十分重要的一环，优化算法受到了大家极大的重视。好的优化算法意味着更好、更快地找到目标模型，而优化算法相关的知识向来都充满了吸引力。深层模型的发展，又让它备受瞩目：由于深层模型的目标函数不再是凸函数，优化曲面变得越来越复杂，别说寻找一个优秀的优化算法，完成正常的优化都变得困难起来。本章将介绍现在被业界广泛了解的一些优化算法，同时简单介绍业界对优化问题的研究情况，同时介绍训练过程中可能遇到的一些问题。

### 8.1 梯度下降是一门手艺活儿

#### 8.1.1 什么是梯度下降法

作为大众耳熟能详的优化算法，**梯度下降法**受到了太多的关注。梯度下降法极易理解，但凡学过数学的读者都有所了解，函数的梯度方向表示了函数值增长速度最快的方向，那么和它相反的方向就可以看作是函数值减少速度最快的方向。对机器学习模型优化的问题，当目标设定为求解目标函数最小值时，只要朝着梯度下降的方向前进，就能不断逼近最优值。

那么梯度下降的算法怎么实现呢？这里给出一种最简单的梯度下降算法——固定学习率的方法，这种梯度下降算法由两个函数和三个变量组成。

- 函数 1：待优化的函数  $f(x)$ ，它可以根据给定的输入返回函数值。
- 函数 2：待优化函数的导数  $g(x)$ ，它可以根据给定的输入返回函数的导数值。



- 变量  $x$ ：保存当前优化过程中的参数值，优化开始时该变量将被初始化成某个数值，优化过程中这个变量会不断变化，直到它找到最小值。
- 变量  $\text{grad}$ ：保存变量  $x$  点处的梯度值。
- 变量  $\text{step}$ ：表示沿着梯度下降方向行进的步长，也被称为学习率（Learning Rate）。它就是本节的主角，在优化中它将固定不变。以下将交替使用步长与学习率两个词，二者实际上是相同的。

包含上面 5 个元素的梯度下降算法如下所示：

```
def gd(x_start, step, g):    # 名称gd代表了Gradient Descent
    x = x_start
    for i in range(20):
        grad = g(x)
        x -= grad * step
        print '[ Epoch {0} ] grad = {1}, x = {2}'.format(i, grad, x)
        if abs(grad) < 1e-6:
            break;
    return x
```

除了上面介绍的内容，代码中还加入了优化的终止条件，由于优化的目标是寻找梯度为 0 的极值点，代码在每一轮迭代结束后衡量变量  $x$  所在的梯度值，因此当梯度值足够小时，就认为  $x$  已经进入最优值附近一个极小的邻域， $x$  和最优值之间的差别不再明显。这时就可以停止优化了。

### 8.1.2 优雅的步伐

明确了算法，下面就通过一个例子展示梯度下降法的优化效果。这里用一个十分简单的函数作为例子：

```
def f(x):
    return x * x - 2 * x + 1

def g(x):
    return 2 * x - 2
```

这个函数  $f(x)$  就是大家在中学常见的二次函数的  $f(x) = x^2 - 2x + 1$ ，读者可以很轻松地看出，最小值是  $x = 1$ ，此时函数值为 0。为了让读者对这个函数有更直观的认识，这里将图画出来：

```
import numpy as np
import matplotlib.pyplot as plt
x = np.linspace(-5,7,100)
y = f(x)
plt.plot(x, y)
```

这个函数如图 8-1 所示。

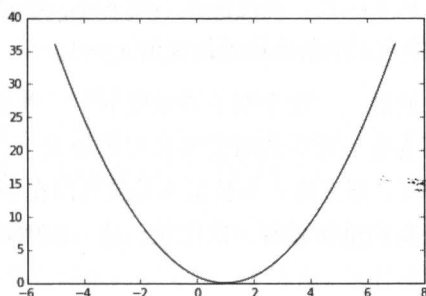


图 8-1 梯度下降法的案例函数

可以清楚地看出这个函数的图像就是一个很简单的抛物线。 $x = 1$  是函数的最小点。下面就用梯度下降法计算它的极值：

```
gd(5,0.1,g)
```

运行后得到了下面的输出：

#以下为运行结果显示

```
[ Epoch 0 ] grad = 8, x = 4.2
[ Epoch 1 ] grad = 6.4, x = 3.56
[ Epoch 2 ] grad = 5.12, x = 3.048
[ Epoch 3 ] grad = 4.096, x = 2.6384
[ Epoch 4 ] grad = 3.2768, x = 2.31072
[ Epoch 5 ] grad = 2.62144, x = 2.048576
[ Epoch 6 ] grad = 2.097152, x = 1.8388608
[ Epoch 7 ] grad = 1.6777216, x = 1.67108864
[ Epoch 8 ] grad = 1.34217728, x = 1.536870912
[ Epoch 9 ] grad = 1.073741824, x = 1.4294967296
[ Epoch 10 ] grad = 0.8589934592, x = 1.34359738368
[ Epoch 11 ] grad = 0.68719476736, x = 1.27487790694
[ Epoch 12 ] grad = 0.549755813888, x = 1.21990232556
[ Epoch 13 ] grad = 0.43980465111, x = 1.17592186044
```

```
[ Epoch 14 ] grad = 0.351843720888, x = 1.14073748836
[ Epoch 15 ] grad = 0.281474976711, x = 1.11258999068
[ Epoch 16 ] grad = 0.225179981369, x = 1.09007199255
[ Epoch 17 ] grad = 0.180143985095, x = 1.07205759404
[ Epoch 18 ] grad = 0.144115188076, x = 1.05764607523
[ Epoch 19 ] grad = 0.115292150461, x = 1.04611686018
```

可以看到，初始值  $x$  从 5 出发，梯度值在不断下降，经过 20 轮迭代， $x$  虽然没有完全等于 1，但是在迭代中它正不断地逼近最优值  $x = 1$ 。

以上就是梯度下降法的展示。这个例子毕竟留下了一点遗憾，最后  $x$  离最优值的距离并不算近，是不是因为每一轮迭代时的步长设置得太小，导致优化值没有更快地收敛到最优值？如果是这样，那么增大步长是不是就可以在 20 轮迭代内看到优化结束呢？抱着这样的想法，让我们重新进行一次优化，这一次的优化步长被设置为之前的 1000 倍（有点夸张）：

```
gd(5,100,g)
```

以下就是步长加大后的结果：

```
#以下为运行结果显示
```

```
[ Epoch 0 ] grad = 8, x = -795
[ Epoch 1 ] grad = -1592, x = 158405
[ Epoch 2 ] grad = 316808, x = -31522395
[ Epoch 3 ] grad = -63044792, x = 6272956805
[ Epoch 4 ] grad = 12545913608, x = -1248318403995
[ Epoch 5 ] grad = -2496636807992, x = 248415362395205
[ Epoch 6 ] grad = 496830724790408, x = -49434657116645595
[ Epoch 7 ] grad = -98869314233291192, x = 9837496766212473605
[ Epoch 8 ] grad = 19674993532424947208, x = -1957661856476282247195
[ Epoch 9 ] grad = -3915323712952564494392, x = 389574709438780167192005
[ Epoch 10 ] grad = 779149418877560334384008, x =
-77525367178317253271208795
[ Epoch 11 ] grad = -155050734356634506542417592, x =
15427548068485133400970550405
[ Epoch 12 ] grad = 30855096136970266801941100808, x =
-3070082065628541546793139530395
[ Epoch 13 ] grad = -6140164131257083093586279060792, x =
610946331060079767811834766548805
```

```
[ Epoch 14 ] grad = 1221892662120159535623669533097608, x =
-121578319880955873794555118543211995
[ Epoch 15 ] grad = -243156639761911747589110237086423992, x =
24194085656310218885116468590099187205
[ Epoch 16 ] grad = 48388171312620437770232937180198374408, x =
-4814623045605733558138177249429738253595
[ Epoch 17 ] grad = -9629246091211467116276354498859476507192, x =
958109986075540978069497272636517912465605
[ Epoch 18 ] grad = 1916219972151081956138994545273035824931208, x =
-190663887229032654635829957254667064580655195
[ Epoch 19 ] grad = -381327774458065309271659914509334129161310392, x =
37942113558577498272530161493678745851550384005
```

可以看到, 参数的梯度不但没有收敛, 反而越来越大。优化的本意是让目标值朝着梯度下降的方向前进, 结果它却走向了另外一个方向。为什么会出现这样的情况呢? 为什么梯度会不降反升呢? 是我们的算法本身有问题, 还是梯度的设置有问题? 解释这个问题还要回到梯度这个概念本身来。

实际上, 函数在某一点的梯度指的是它在当前变量处的梯度, 对于这一点来说, 它的梯度方向指向了函数上升的方向, 可以利用泰勒公式证明在一定的范围内, 沿着负梯度方向前进, 函数值是会下降的。但是, 公式只能保证在一定范围内是成立的, 从函数的实际图像中也可以看出, 如果优化的步长太大, 就有可能跳出函数值下降的范围, 那么函数值是否下降就不好说了, 当然有可能会越变越大, 造成优化的悲剧。

如何避免这种悲剧发生呢? 简单的方法就是将步长减少, 像前面的实验那样设得小一些。当然还有一些 Line-search 的方法可以通过其他限定条件避免这样的事情发生, 这里就不做介绍了。

既然理论没有错, 那么看起来只能通过修改步长来完成这个问题了。既然小步长会使目标值的梯度下降, 大步长会使梯度发散, 那么有没有一个步长会让优化问题原地打转呢? 在这个问题中, 这样的步长是存在且容易找到的。

由于梯度优化总会造成数值的改变, 所以每一步优化都让目标值原地打转是不太现实的, 这里可以假设目标值从  $A$  变更到  $B$ , 再从  $B$  变更到  $A$ , 如此循环更新。于是这里可以假设: 从  $x = 5$  出发, 经过一轮迭代,  $x$  被更新到了另一个值  $x'$ , 再用  $x'$  继续迭代,  $x'$  又更新到了 5。上述过程可以写成一个方程:

$$x = 5, g(x) = 8, \text{新的值 } x' = 5 - 8\text{step}$$

$$g(x') = 2 \cdot (5 - 8\text{step}) - 2, \text{回到过去: } x' - g(x')\text{step} = x = 5$$

$$\text{合并公式求解得, } \text{step} = 1$$

也就是说  $step = 1$  时，求解会原地打转，那么接下来就来试一下：

```
gd(5,1,g)
```

它的结果如下所示：

```
#以下为运行结果显示
```

```
[ Epoch 0 ] grad = 8, x = -3
[ Epoch 1 ] grad = -8, x = 5
[ Epoch 2 ] grad = 8, x = -3
[ Epoch 3 ] grad = -8, x = 5
[ Epoch 4 ] grad = 8, x = -3
[ Epoch 5 ] grad = -8, x = 5
[ Epoch 6 ] grad = 8, x = -3
[ Epoch 7 ] grad = -8, x = 5
[ Epoch 8 ] grad = 8, x = -3
[ Epoch 9 ] grad = -8, x = 5
[ Epoch 10 ] grad = 8, x = -3
```

和预想的一样，目标值在函数中打转，梯度下降法失效了。

通过上面的实验可以发现，对于初始值为 5 这个点，当步长大于 1 时，梯度下降法会出现求解目标值发散的现象，而小于 1 则不会发散，参数会逐渐收敛。那么 1 就是步长的临界点。那么问题又来了，对于别的初始值，这个规律还适用吗？接下来就把初始值换成 4，再进行一次实验。

```
gd(4,1,g)
```

```
#以下为运行结果显示
```

```
[ Epoch 0 ] grad = 6, x = -2
[ Epoch 1 ] grad = -6, x = 4
[ Epoch 2 ] grad = 6, x = -2
[ Epoch 3 ] grad = -6, x = 4
[ Epoch 4 ] grad = 6, x = -2
[ Epoch 5 ] grad = -6, x = 4
[ Epoch 6 ] grad = 6, x = -2
[ Epoch 7 ] grad = -6, x = 4
[ Epoch 8 ] grad = 6, x = -2
[ Epoch 9 ] grad = -6, x = 4
[ Epoch 10 ] grad = 6, x = -2
```

实验发现，步长等于 1 时，无论初始值设置成（最优值除外的）任何数字，参数都不会收敛到最优值。这个实验揭示了一个道理：对于这个二次函数，如果采用固定步长的梯度下降法进行优化，步长要小于 1，否则不论初始值等于多少，问题都会发散或者原地打转！

如果再换一个函数： $4x^2 - 4x + 1$ ，它的安全步长是多少呢？还是 1 吗？当然不是。这一次这个值是 0.25，相应的实验如下所示：

```
def f2(x):
    return 4 * x * x - 4 * x + 1
def g2(x):
    return 8 * x - 4
gd(5,0.25,g2)
```

#以下为运行结果显示

```
[ Epoch 0 ] grad = 36, x = -4.0
[ Epoch 1 ] grad = -36.0, x = 5.0
[ Epoch 2 ] grad = 36.0, x = -4.0
[ Epoch 3 ] grad = -36.0, x = 5.0
[ Epoch 4 ] grad = 36.0, x = -4.0
[ Epoch 5 ] grad = -36.0, x = 5.0
[ Epoch 6 ] grad = 36.0, x = -4.0
[ Epoch 7 ] grad = -36.0, x = 5.0
[ Epoch 8 ] grad = 36.0, x = -4.0
[ Epoch 9 ] grad = -36.0, x = 5.0
[ Epoch 10 ] grad = 36.0, x = -4.0
```

实验结果和猜测完全一致。到此为止，相信读者对步长的设置有了更深的认识。这组实验说明了梯度下降法简单中的不简单，虽然固定步长的方法看上去简单直观，可是步长的选择需要慎重。即使这样一个一维的二次函数都存在优化问题，对于本书关注的 CNN 网络优化来说，优化曲面比上面的二次函数复杂很多。采用相对固定步长优化实际上是“步步惊心”。实战中需要一定的尝试与技巧才能找到最合适的步长。

## 8.2 路遥知马力：动量

本节将介绍动量（Momentum）算法<sup>[1]</sup>。动量是物理课上学习过的一个概念，正如它的中文名一样，在优化求解的过程中，动量代表了之前迭代优化量，它将在后面的

优化过程中持续发威，推动目标值前进。拥有了动量，一个已经结束的更新量不会立刻消失，只会以一定的形式衰减，剩下的能量将继续在优化中发挥作用。

上面介绍的概念比较抽象，以下就是基于动量的梯度下降的代码，这个代码基于前面的梯度下降法做了一定的修改：

```
def momentum(x_start, step, g, discount = 0.7):
    x = np.array(x_start, dtype='float64')
    pre_grad = np.zeros_like(x)
    for i in range(50):
        grad = g(x)
        pre_grad = pre_grad * discount + grad * step
        x -= pre_grad
        print '[ Epoch {0} ] grad = {1}, x = {2}'.format(i, grad, x)
        if abs(sum(grad)) < 1e-6:
            break;
    return x
```

代码中多出了一个新变量 *pre\_grad*。这个变量就是用于存储历史积累的动量，每一轮迭代动量都会乘以一个打折量 (*discount*) 做能量衰减，但是它依然会被用于更新参数。

每个事物的存在必然有它的道理，动量算法和前面介绍的梯度下降法相比有什么优点呢？用形象的话来说，它可以帮助目标值穿越“狭窄山谷”形状的优化曲面，从而到达最终的最优点。那么，什么是“山谷”，怎么理解“穿越山谷”这个词语呢？下面用一个例子来解释。这次的问题和 8.1 节相比复杂些，是一个二元二次函数： $z = x^2 + 50y^2$ 。

```
def f(x):
    return x[0] * x[0] + 50 * x[1] * x[1]
def g(x):
    return np.array([2 * x[0], 100 * x[1]])
xi = np.linspace(-200,200,1000)
yi = np.linspace(-100,100,1000)
X,Y = np.meshgrid(xi, yi)
Z = X * X + 50 * Y * Y
```

函数在等高线图上的样子如图 8-2 所示。

其中中心的点表示了最优值。把等高线上的图像想象成地形图，从等高线的疏密程度可以看出，这个函数在 *y* 轴方向十分陡峭，在 *x* 轴方向则相对平缓。也就是说，函

数在  $y$  轴的方向导数比较大, 在  $x$  轴的方向导数比较小。图 8-2 所示的区域可以看作是一个“山谷”。如果用 8.1 节介绍的固定步长的梯度下降法尝试优化, 那么有:

```
gd([150,75], 0.016, g)
```

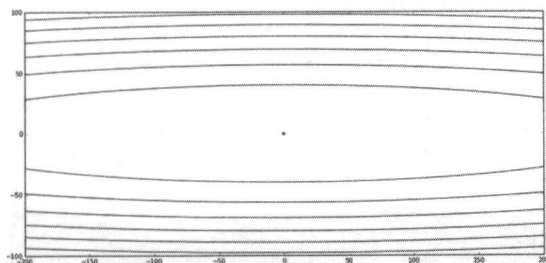


图 8-2 二次目标函数的等高线图

这里将 50 轮迭代过程中参数的优化过程图画出来, 如图 8-3 所示。

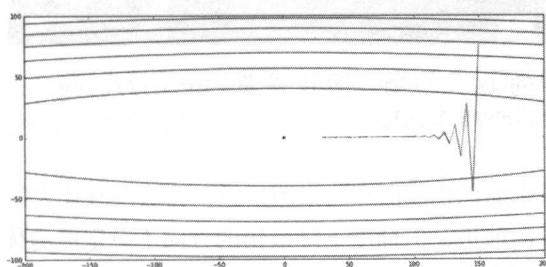


图 8-3 梯度下降法的优化曲线

可以看出目标值从某个点出发, 整体趋势向着最优点前进, 它和最优值的距离不断靠近, 说明优化过程在收敛, 步长设置是没有问题的, 但是前进的速度似乎有点乏力, 50 轮迭代并没有到达最优值。直觉上认为, 有可能步长设置得偏小。有 8.1 节的经历, 这次设置步长变得小心了许多。我们只将步长稍微变大一些, 结果如图 8-4 所示。

```
res, x_arr = gd([150,75], 0.019, g)
contour(X,Y,Z, x_arr)
```

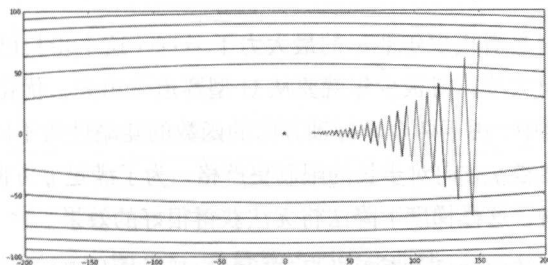


图 8-4 增大步长的梯度下降法优化曲线



虽然优化效果有了一定的进步，但成效依然不明显，而且优化的过程图中出现了参数值左右抖动的现象。这个现象是怎么回事呢？看上去参数在优化过程中产生了某种“打转”的现象，做了很多无用功。看到这个曲线路径，读者可能会想到一个比较熟悉的极限运动赛道，如图 8-5 所示。

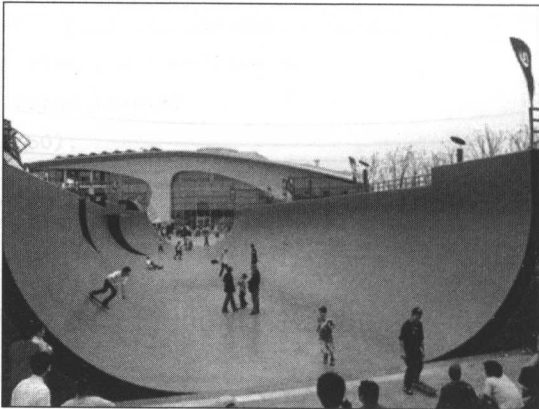


图 8-5 极限运动中的 U 型赛道（图片来源：[http://blog.sina.com.cn/s/blog\\_61aecc840100rb9b.html](http://blog.sina.com.cn/s/blog_61aecc840100rb9b.html)）

实际上，算法眼中的优化曲面和这张图很像，算法没有让大家“失望”，一直在选择梯度下降最快的方向前进，但梯度下降最快的方向不一定能为优化提供太多的帮助，因此沿着这条道路进行优化就十分艰难——就像这个极限运动一样，从一边的高台滑下，然后滑到另一边，不断进行。这个过程很像 8.1 节介绍的“打转”现象，朝正确优化方法前进的步伐比较小，而无效重复的前进步伐比较大。虽然优化的效率很低，但这就是梯度下降法。它的眼中只有梯度，并且只相信梯度。

如果继续增加步长，优化曲线会变成什么样子呢？“滑板少年”还能不能再快点前进呢？实验结果如图 8-6 所示。

```
res, x_arr = gd([150,75], 0.02, g)
contour(X,Y,Z, x_arr)
```

从结果来看，这是滑板少年能尽的最大力了……这个步长已经是能设置的最大步长，如果步长再设大些，滑板少年就要从 U 型赛道飞出去，优化参数的梯度也将发散出去。在这个问题中，由于两个坐标轴方向的函数的陡峭性质不同，两个方向对最大步长的限制不同，显然  $y$  方向对步长的限制更严格。为了满足  $y$  方向的更新， $x$  方向就无法获得充分的更新，这样梯度下降法将无法获得很好的效果。

下面我们将目光转向动量，看看动量算法如何帮助这位滑板少年。

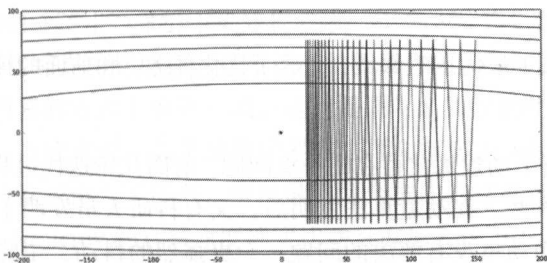


图 8-6 极限步长下梯度下降法的优化曲线

在开始动量算法计算前，首先要对滑板少年的行动方向做一个总结，读者可以发现滑板少年每一次的行动只会在以下三个方向进行。

- 沿  $-x$  方向滑行
- 沿  $+y$  方向滑行
- 沿  $-y$  方向滑行

这样看来，要是少年能把行动的力量集中在往  $-x$  方向走而不是在  $y$  方向打转就好了。于是这个想法将分为两个部分。

1. 集中力量沿  $-x$  方向走。
2. 尽量不要在  $y$  轴打转。

这两个想法可以被动量算法实现。我们可以想象，当使用了动量后，历史的更新量会以衰减的形式不断作用在这些方向上，那么沿  $-y$  和  $+y$  两个方向的动量就可以相互抵消，而  $-x$  方向的力则会一直加强，这样滑板少年虽然还会在  $y$  方向打转，但是他在  $-x$  方向的速度会因为之前的累积而变得越来越快。

基于上面的分析，下面来看看加了动量技能的滑板少年的优化表现图，如图 8-7 所示。

`momentum([150,75], 0.016, g)`

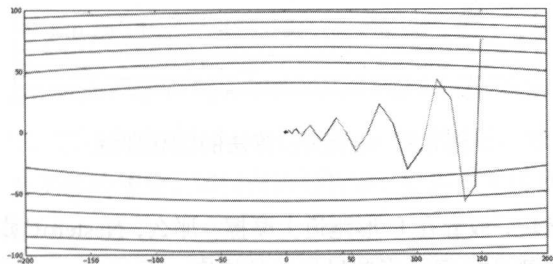


图 8-7 动量算法的优化曲线

优化曲线的果然没有令人失望，尽管滑板少年还是产生了一些打转，但是在 50 轮迭代后，他还是进入了最优点的邻域，完成了优化任务，和前面的梯度下降法相比有了很大的进步。

当然，还是暴露了动量优化存在的一点问题，前面几轮迭代过程中目标值在  $y$  轴上的震荡比过去还要大些。在动量算法发明后，又有科研人员发明了基于动量算法的改进算法，解决了前面动量没有解决的问题——更强烈的抖动，干脆让滑板少年停止玩耍，专心赶路。这就是 Nesterov 算法 [2]。这里给出代码和刚才问题的优化过程，如图 8-8 所示。

```
def nesterov(x_start, step, g, discount = 0.7):
    x = np.array(x_start, dtype='float64')
    pre_grad = np.zeros_like(x)
    for i in range(50):
        x_future = x - step * discount * pre_grad
        grad = g(x_future)
        pre_grad = pre_grad * 0.7 + grad
        x -= pre_grad * step

        print '[ Epoch {0} ] grad = {1}, x = {2}'.format(i, grad, x)
        if abs(sum(grad)) < 1e-6:
            break;
    return x

nesterov([150,75], 0.012, g)
```

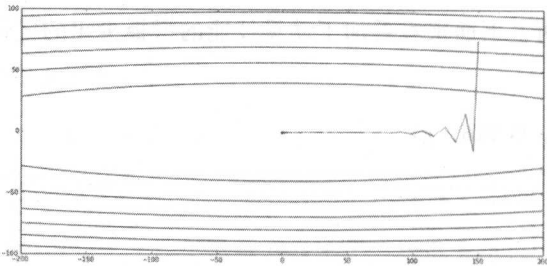


图 8-8 Nesterov 算法的优化曲线

滑板少年不再贪玩，放弃在 U 形跑道上摩擦。那么，Nesterov 算法和动量算法相比有什么区别呢？动量算法计算了当前目标点的梯度；而 Nesterov 算法计算了动量更新后优化点的梯度。这其中关键的区别在于计算梯度的点。

读者可以想象一个场景，当优化点已经积累了某个抖动方向的梯度后，这时对于动量算法来说，虽然当前点的梯度指向积累梯度的相反方向，但是量不够大，所以最终的优化方向还会在积累的方向上前进一段，这就是图 8-8 所示的优化步骤图所展示的效果。对 Nesterov 方法来说，如果按照积累方法再向前多走一段，这时梯度中指向积累梯度相反方向的量变得大了许多，所以最终两个方向的梯度抵消，反而使得抖动方向的量迅速减少。Nesterov 的衰减速度确实比动量方法要快不少。

经过上面的介绍，相信读者能够理解动量算法在“穿越山谷”上的卓越表现。本节最后还要讲述动量在数值上的事情。很多科研人员已经给出了动量打折率的建议配置——0.9（刚才的例子全部是 0.7），那么 0.9 的动量打折率能使历史更新量发挥多大作用呢？

如果用  $\mathbf{G}$  表示每一轮迭代的动量， $g$  表示当前一轮迭代的更新量（方向  $\times$  步长）， $t$  表示迭代轮数， $\gamma$  表示动量的打折率，那么对于时刻  $t$  的梯度更新量就有如下的公式：

$$\mathbf{G}_t = \gamma \mathbf{G}_{t-1} + g_t$$

$$\mathbf{G}_t = \gamma(\gamma \mathbf{G}_{t-2} + g_{t-1}) + g_t$$

$$\mathbf{G}_t = \gamma^2 \mathbf{G}_{t-2} + \gamma g_{t-1} + g_t$$

$$\mathbf{G}_t = \gamma^t g_0 + \gamma^{t-1} g_1 + \cdots + g_t$$

这样可以计算出对于第一轮迭代的更新  $g_0$  来说，从  $\mathbf{G}_0$  到  $\mathbf{G}_T$ ，它的总贡献量为

$$(\gamma^t + \gamma^{t-1} + \cdots + \gamma + 1)g_0$$

相信读者已经发现它的贡献和是一个等比数列的和，比值为  $\gamma$ 。如果  $\gamma = 0.9$ ，那么根据公比小于 1 的等比数列的极限公式，可以知道更新量在极限状态下贡献值：

$$\lim_{t \rightarrow \infty} (\gamma^t + \gamma^{t-1} + \cdots + \gamma + 1)g_0 = \frac{g_0}{1 - \gamma}$$

那么当  $\gamma = 0.9$  时，它一共贡献了相当于自身 10 倍的能量。如果  $\gamma = 0.99$  呢？那就是 100 倍能量了。在实际应用中，如何设置打折率，就需要分析具体任务中更新量需要多么“持久”的动力了。

### 8.3 SGD 的变种算法

本节将介绍深度学习中其他的优化算法，这些算法出现的时间比较短，但是它们都在一些应用中证明了自己的优化能力。

#### 8.3.1 非凸函数

前面讲解的例子中展示的函数都是凸函数，而在深层模型中，非凸函数才是最常见的优化曲面，因此本节将以一个非凸函数为例展开：

$$f(x,y) = x^2 - 2x + 100xy + 10y^2 + 20y$$

函数在三维图像的样子并不容易观察，因此这里直接画出它的等高线图——它的样子如图 8-9 所示。

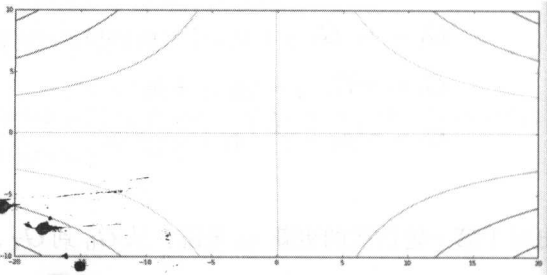


图 8-9 目标函数在二维等高线上的样子

在图 8-9 中，第一、第三象限的函数值偏大，第二、第四象限的函数值偏小。为了找到最小值，参数应该向着第二、第四象限前进。从图像中就可以看出，这个函数不满足凸函数的性质：函数不满足 Jensen 不等式，函数的 Hessian 矩阵不是正定矩阵等，接下来介绍的优化算法就要在这个函数上进行优化。

每种优化方法都要在这个函数上做两个实验。

第一个实验是“弯道赛”，目标值从一个比较正常的位置——(5,5) 点出发，看看各种算法的参数更新轨迹。

第二个实验是“爬坡赛”，目标值从鞍点附近出发，看看各种算法在不借助历史信息的情况下如何冲出鞍点。

在这个新的优化曲面上，前几节介绍的经典算法——梯度下降、动量算法也将出场参与实验。在进行第一个实验的同时，我们将介绍这些新算法，之后进行第二个实验。

### 8.3.2 经典算法的弯道表现

首先是梯度下降法，这里直接给出执行代码和优化轨迹，如图 8-10 所示。

```
res, x_arr = gd([5,5], 0.0008, g)
contour(X,Y,Z, x_arr)
```

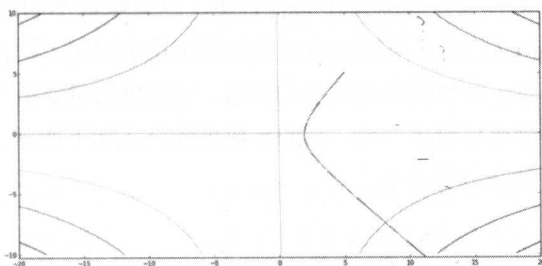


图 8-10 梯度下降的优化曲线

为了保证算法经过 50 轮迭代不会超出图像的范围，这里对算法的学习率进行了限制。可以看出梯度下降法在优化过程中首先冲向了局部最优点，同时也是函数的鞍点，在行进过程中梯度值随着曲线迅速改变，于是它调转头冲向了真正的优化点。可见梯度下降法对梯度的瞬时改变十分灵敏。

然后是动量算法，优化轨迹图如图 8-11 所示。

```
res, x_arr = momentum([5, 5], 3e-4, g)
contour(X,Y,Z, x_arr)
```

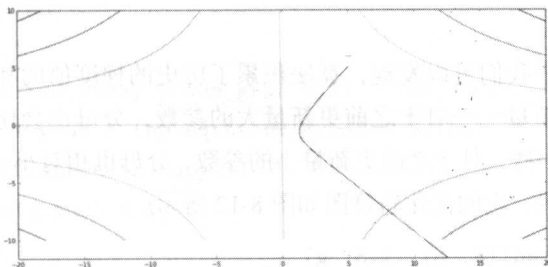


图 8-11 动量算法的优化曲线

可以看出动量算法也成功地绕过了鞍点，不过由于累积的更新量的原因，动量算法“转身”的时机相对晚一些。同时动量算法的学习率设置比梯度下降还要小。

### 8.3.3 Adagrad

**Adagrad**<sup>[3]</sup> 是一种自适应的梯度下降方法，何为自适应呢？在梯度下降法中，参数的更新量等于梯度乘以学习率，也就是说，更新量和梯度是线性正相关的；而在实际应用中，每个参数的梯度各有不同，有的梯度比较大，有的比较小，那么就有可能遇到参数优化不均衡的情况。

参数优化不均衡对模型训练来说不是件好事，这意味着不同的参数更新适用于不同的学习率。而 Adagrad 的自适应算法也正是要解决这个问题。算法希望不同参数的更新量能够比较均衡。对于已经更新比较多的参数，它的更新量要适当衰减，而更新比较少的参数，它的更新量要尽量多一些，它的参数更新公式如下所示，其中  $\epsilon$  的取值一般比较小，它只是为了防止分母为 0：

$$x_{-} = \text{lr} \cdot \frac{g}{\sqrt{\sum g^2 + \epsilon}}$$

它的核心代码如下所示：

```
def adagrad(x_start, step, g, delta=1e-8):
    x = np.array(x_start, dtype='float64')
    sum_grad = np.zeros_like(x)
    for i in range(50):
        grad = g(x)
        sum_grad += grad * grad
        x -= step * grad / (np.sqrt(sum_grad) + delta)
        if abs(sum_grad) < 1e-6:
            break;
    return x
```

从公式和代码中我们可以发现，算法积累了历史的梯度值的和，并用这个加和来调整每个参数的更新量——对于之前更新量大的参数，分母也会比较大，于是未来它的更新量会相对小一些；对于之前更新量小的参数，分母也相对小一些，于是未来它的更新量会相对大一些。它的优化轨迹图如图 8-12 所示。

```
res, x_arr = adagrad([5, 5], 1.3, g)
contour(X,Y,Z, x_arr)
```

从总体效果上看，除了学习率比较大，这和前面的算法没什么差别。除此之外，Adagrad 的优化路径和前面的算法相比更靠近鞍点，由于算法开始时，分母的数值相对较小，因此更新量比较大。

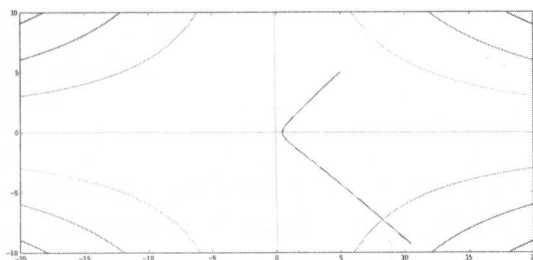


图 8-12 Adagrad 算法的优化曲线

### 8.3.4 Rmsprop

前面的 Adagrad 算法有一个很大的问题，那就是随着优化的迭代次数不断增加，更新公式的分母项会变得越来越大。所以理论上更新量也会越来越小，这对优化十分不利。下面的算法 **Rmsprop**<sup>[4]</sup> 就试图解决这个问题。在它的算法中，分母的梯度平方不再随优化而递增，而是做加权平均。更新公式如下所示：

$$G_{t+1} = \beta G_t + (1 - \beta)g^2$$

$$x_{t+1} = x_t - \text{lr} \frac{g}{\sqrt{G_{t+1} + \varepsilon}}$$

它的代码也比较简单，如下所示：

```
def rmsprop(x_start, step, g, rms_decay = 0.9, delta=1e-8):
    x = np.array(x_start, dtype='float64')
    sum_grad = np.zeros_like(x)
    passing_dot = [x.copy()]
    for i in range(50):
        grad = g(x)
        sum_grad = rms_decay * sum_grad + (1 - rms_decay) * grad * grad
        x -= step * grad / (np.sqrt(sum_grad) + delta)
        passing_dot.append(x.copy())
        if abs(sum(grad)) < 1e-6:
            break;
    return x, passing_dot
```

下面同样来看看它的表现，优化轨迹图如图 8-13 所示。

```
res, x_arr = rmsprop([5, 5], 0.3, g)
contour(X,Y,Z, x_arr)
```



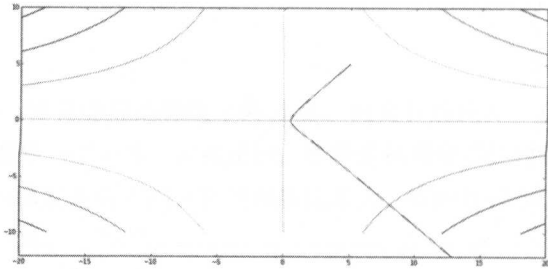


图 8-13 RmsProp 算法优化曲线

看上去和前面 Adagrad 的表现还是差不多的，但是它的学习率比 Adagrad 要小很多，而更新的速度也比 Adagrad 快，这说明分母对优化的阻碍明显变小了。

8.3.5 AdaDelta

Adagrad 和 Rmsprop 算法已经在某种程度上解决了“自适应确定更新量”的问题，但是在 AdaDelta 算法<sup>[5]</sup>的眼中似乎还不够。在介绍它的论文中，作者详细阐述了 Adagrad 算法中得到的参数更新量的“单位”不对。

在之前的一些优化算法中，更新量都是由学习率乘以梯度向量组成，而 Adagrad 方法在更新量计算的公式中除以了梯度累积量，这相当于打破了之前的更新量组成部分的平衡性，因此算法的作者认为如果分母加上了梯度累积量，那么分子也应该加上一些内容，这样的更新量才会和之前的算法一样保持平衡。更新量的“单位”才能恢复正常。

于是作者基于 Adagrad 进行了修改，就有了下面的计算公式：

$$\begin{aligned} G_{t+1} &= \beta G_t + (1 - \beta)g^2 \\ \delta_{t+1} &= \sqrt{\frac{\Delta_t + \varepsilon}{G_{t+1} + \varepsilon}}g \\ x_{t+1} &= x_t - \text{lr}\delta_{t+1} \\ \Delta_{t+1} &= \beta\Delta_t + (1 - \beta)\delta_{t+1}^2 \end{aligned}$$

具体的代码如下所示：

```
def adadelata(x_start, step, g, momentum = 0.9, delta=1e-1):
    x = np.array(x_start, dtype='float64')
    sum_grad = np.zeros_like(x)
```

```

sum_diff = np.zeros_like(x)
passing_dot = [x.copy()]
for i in range(50):
    grad = g(x)
    sum_grad = momentum * sum_grad + (1 - momentum) * grad * grad
    diff = np.sqrt((sum_diff + delta) / (sum_grad + delta)) * grad
    x -= step * diff
    sum_diff = momentum * sum_diff + (1 - momentum) * (diff * diff)
    passing_dot.append(x.copy())
    if abs(sum(grad)) < 1e-6:
        break;
return x, passing_dot

```

算法的优化轨迹如图 8-14 所示。

```

res, x_arr = adadelta([5, 5], 0.4, g)
contour(X,Y,Z, x_arr)

```

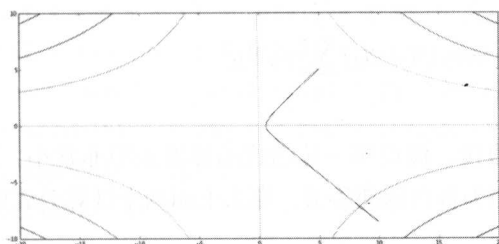


图 8-14 AdaDelta 算法优化曲线

和 Adagrad 相比，它的学习率小了一些，但是和 RmsProp 相比，两者的学习率比较相近，说明两个算法选择了不同的角度修改 Adagrad 算法，最终的效果比较相似。

### 8.3.6 Adam

Adam 算法<sup>[6]</sup>可以算作是前面两类算法的集大成者。它既包含了动量算法的思想，也包含了 RmsProp 的自适应梯度的思想。在计算过程中，Adam 既要像动量算法那样计算累积的动量：

$$m_{t+1} = \beta_1 m_t + (1 - \beta_1) g_t$$

又要像 RmsProp 那样计算梯度的滑动平方和：

$$v_{t+1} = \beta_2 v_t + (1 - \beta_2) g_t^2$$

作者没有直接把这两个计算值加入最终计算的公式中。作者希望  $m_{t+1}$  能够等于到  $t$  时刻时梯度平均值，这样这个数值可以用来近似的梯度期望值  $E[g]$ ；同时  $v_{t+1}$  能够等于到  $t$  时刻为止梯度平方的平均值，这样这个数值就可以用来近似梯度的二阶矩  $E[g^2]$ 。但是很显然，滑动平均的计算方法和直接求平均的方法存在差异，为此作者希望通过简单的方法纠正其中的差异。将上面的两个积累量展开，可以得到：

$$\begin{aligned} m_{t+1} &= \beta_1(\beta_1 m_{t-1} + (1 - \beta_1) g_{t-1}) + (1 - \beta_1) g_t \\ &= \beta_1^2 m_{t-1} + (1 - \beta_1)[\beta_1 g_{t-1} + g_t] \\ &= (1 - \beta_1) \sum_{i=1}^t \beta_1^{t-i} g_i \\ v_{t+1} &= \beta_2(\beta_2 v_{t-1} + (1 - \beta_2) g_{t-1}^2) + (1 - \beta_2) g_t^2 \\ &= \beta_2^2 v_{t-1} + (1 - \beta_2)[\beta_2 g_{t-1}^2 + g_t^2] \\ &= (1 - \beta_2) \sum_{i=1}^t \beta_2^{t-i} g_i^2 \end{aligned}$$

这里需要做一个假设，假设每一轮迭代的梯度差距非常小，这样公式中的梯度和梯度的平方就可以近似为各自的期望值，那么上面两个计算公式就变成了：

$$\begin{aligned} m_{t+1} &\simeq (1 - \beta_1) \sum_{i=1}^t \beta_1^{t-i} E[g] \\ &= (1 - \beta_1) \times \frac{1 - \beta_1^t}{1 - \beta_1} E[g] \\ &= (1 - \beta_1^t) E[g] \\ v_{t+1} &\simeq (1 - \beta_2) \sum_{i=1}^t \beta_2^{t-i} E[g_i^2] \\ &= (1 - \beta_2) \times \frac{1 - \beta_2^t}{1 - \beta_2} E[g_i^2] \\ &= (1 - \beta_2^t) E[g_i^2] \end{aligned}$$

得到了这两个推导公式，作者就发现了两个计算量与期望的差距，于是给这两个变量加上了这个修正量，两个修正后的计算量变成：

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

到这时，两个计算量将被融合到最终的公式中：

$$x_{t+1} = x_t - \text{lr} \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \varepsilon}$$

它的代码如下所示：

```
def adam(x_start, step, g, beta1 = 0.9, beta2 = 0.999, delta=1e-8):
    x = np.array(x_start, dtype='float64')
    sum_m = np.zeros_like(x)
    sum_v = np.zeros_like(x)
    passing_dot = [x.copy()]
    for i in range(50):
        grad = g(x)
        sum_m = beta1 * sum_m + (1 - beta1) * grad
        sum_v = beta2 * sum_v + (1 - beta2) * grad * grad
        correction = np.sqrt(1 - beta2 ** i) / (1 - beta1 ** i)
        x -= step * correction * sum_m / (np.sqrt(sum_v) + delta)
        passing_dot.append(x.copy())
        if abs(sum(grad)) < 1e-6:
            break;
    return x, passing_dot
```

它的优化轨迹图如 8-15 所示。

```
res, x_arr = adam([5, 5], 0.1, g)
contour(X,Y,Z, x_arr)
```

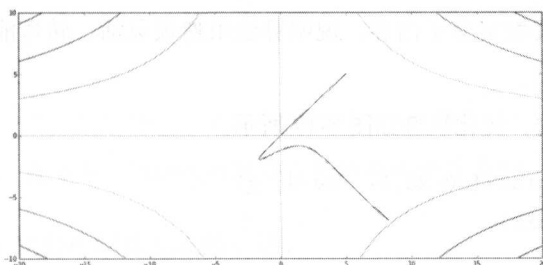


图 8-15 Adam 算法优化曲线

因为 Adam 算法结合了冲量的“惯性”和自适应算法“起步快”这两个特点，所以它是唯一一个让目标点冲过鞍点的算法。当然最后目标点还是绕了回来。

以上就是这些算法的介绍，以及它们在“拐弯赛”中的表现。前面提到这个实验检验了算法的“拐弯”能力。如果一个优化的问题的优化曲面十分复杂，上面充满了各种崎岖坎坷的山峰和山谷，那么目标点就需要经常适应这种环境并快速做出反应。现在每一个算法都给读者交出了答案。仔细看来每个算法的表示还是不太一样的。有的算法比较灵敏，一旦发现新的优化方向就会选择“转身”；有的算法则是身体比较“笨重”，直到“冲过了头才转过身来”；有的算法转弯很轻松，并不需要设置很大的学习率；有的算法则需要设置较大的学习率才能完成转弯，否则就会行动缓慢。

### 8.3.7 爬坡赛

转弯赛到此结束，下面进入另一个比赛——爬坡赛。爬坡赛的比赛规则是，所有算法使用和第一轮比赛同样的参数，从鞍点附近出发，经过 50 轮迭代，看看它能走到哪里。

首先是老朋友梯度下降法，它的优化轨迹如图 8-16 所示。

```
res, x_arr = gd([-0.23, 0.0], 0.0008, g)
contour(X, Y, Z, x_arr)
```

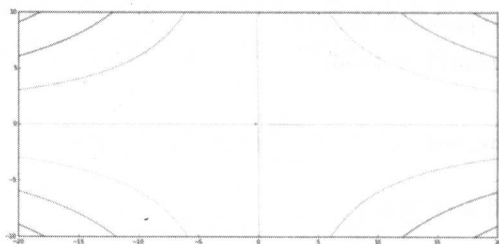


图 8-16 梯度下降法在爬坡赛的优化曲线

梯度下降法在爬坡赛交了白卷，说明算法虽然很灵活，转弯能力很强，但是动力不足。

下面是动量算法，优化轨迹如图 8-17 所示。

```
res, x_arr = momentum([-0.23, 0], 5e-4, g)
contour(X, Y, Z, x_arr)
```

动量算法比梯度下降法好一点，但是好得有限。

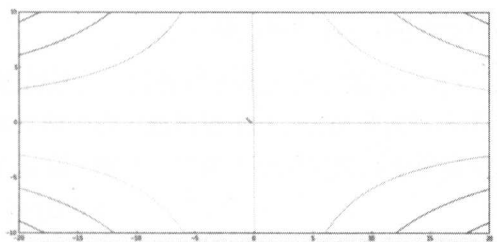


图 8-17 动量算法在爬坡赛的优化曲线

然后是 Adagrad，优化轨迹如图 8-18 所示：

```
res, x_arr = adagrad([-0.23, 0], 1.3, g)
contour(X,Y,Z, x_arr)
```

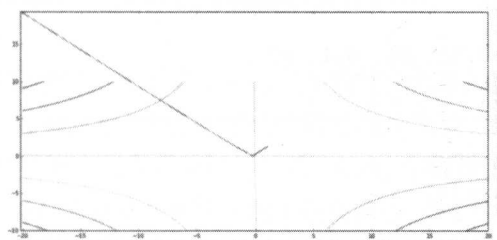


图 8-18 Adagrad 算法在爬坡赛的优化曲线

可以看出它一改弯道赛中平平的表现，很轻松地冲向了最优值。

下面是 RmsProp，它的优化轨迹如图 8-19 所示。

```
res, x_arr = rmsprop([-0.23, 0], 0.3, g)
contour(X,Y,Z, x_arr)
```

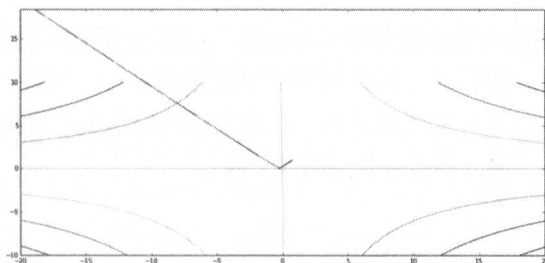


图 8-19 RmsProp 算法在爬坡赛的优化曲线

RmsProp 在这个比赛中同样完成的不错。

下面我们再看 AdaDelta，它的优化轨迹如图 8-20 所示。

```
res, x_arr = adadelta([-0.23, 0], 0.4, g)
contour(X,Y,Z, x_arr)
```

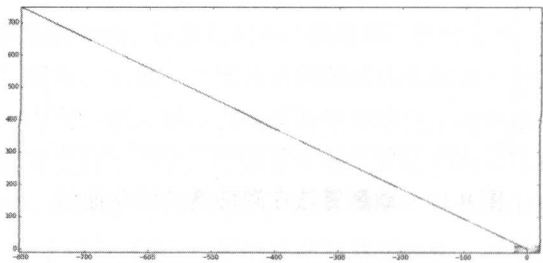


图 8-20 AdaDelta 算法在爬坡赛的优化曲线

可以看到 AdaDelta 算法一骑绝尘冲向了最优值，而且目标值前进的速度越来越快。最后登场的是 Adam，它的优化轨迹如图 8-21 所示。

```
res, x_arr = adam([-0.23, 0], 0.1, g)
contour(X,Y,Z, x_arr)
```

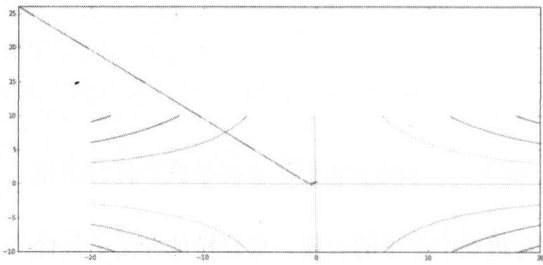


图 8-21 Adam 算法在爬坡赛的优化曲线

它的目标值同样是跑出了原始规划的区域，而且每一步的更新量比较相近。

看完了所有算法的表现，这里对所有算法的表现做一个总结。如果说弯道赛考查了算法的“敏捷”，那么爬坡赛就考察了算法的“力量”。在爬坡赛中，基于自适应机制的算法普遍表现优异，说明这些算法确实可以做到自适应：当优化函数提供的梯度较小时，它同样能提供足够的参数更新量，而非自适应的算法对优化曲面的环境比较敏感，如果梯度比较小，那么更新量就会明显下降。

8.3.8 总结

本节经过两轮的实验——弯道赛和爬坡赛，测试了几种算法在同一配置上的实力。实际上，测试的目的不是比较哪个算法在哪个实验上更厉害，而是比较另外一个实力——哪个算法同时在两个实验上表现稳定。

在实际使用过程中，在一段时间内学习率一般是固定的，在优化过程中无法预料目标点会碰上什么样的情景，它的梯度会是多少，那么读者肯定希望自己使用的算法又能转弯又能爬坡，而且两个能力越相近越好，这样设置学习率也越轻松。如果一个算法转弯厉害但是爬坡比较弱，那么一旦遇上爬坡它就会走得很慢，反之也是如此，这样的话设置学习率就容易顾此失彼。

所以从刚才的表现看，RmsProp 和 Adam 表现得更平稳，现在也确实有越来越多的科研人员在他们的论文中选择这两种优化方法。当然，对这些算法效果的研究主要还停留在实验层面。希望有一天科研人员能够从理论上找出最优秀的优化算法。

## 8.4 L1 正则的效果

本节将把目光转向正则化，来看看正则化对参数优化结果的影响。在刚开始学习机器学习时，相信读者也接触过很多概念：经验风险和结构风险、bias and variance、过拟合等。这些概念和参数正则化都有着密切的关系。Caffe 中提供了两种正则化方法：L2 正则和 L1 正则，它们也是大家最耳熟能详的正则化算法。L2 的优化目标公式如下：

$$\text{Obj}(w) = \text{Loss}(w) + \frac{1}{2}\lambda \sum_i w_i^2$$

L1 正则的优化目标公式是这样的：

$$\text{Obj}(w) = \text{Loss}(w) + \lambda \sum_i |w_i|$$

它们有什么区别呢？在许多经典的机器学习课本中经常用如图 8-22 所示的图来教导我们。

这张图展示了损失函数和正则函数之间的关系。从图中可以看出，假设损失函数的主体是一个凸函数，它的等高线均匀地向外扩散。在正方形的 L1 正则约束下，目标函数的最优值更容易出现在坐标轴上。这样的参数在有些坐标轴上为 0，因此最优参数也就具有稀疏性。而圆形的 L2 正则就不太容易达到这个效果，从图上看最优参数不会落在坐标轴上，因此它也不容易获得稀疏性。所以多年来，相信读者也一直在心中默默记住：针对 L1 正则优化可以达到参数稀疏化的效果。而且，在很多经典的机器学习模型中，L1 正则带来的稀疏效果还是不错的。当然，理论上依然存在使用 L1 正则却无法达到稀疏效果的情况，这个情况比较特殊，这里就不细说了。



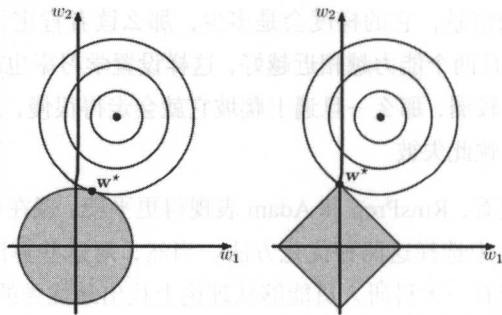


图 8-22 L1 和 L2 的正则优化原理示意图（图片来源：Bishop C M. Pattern Recognition and Machine Learning[M]. Springer-Verlag New York, Inc. 2006.）

与 L1 正则相关的优化算法比较多也比较复杂，本书中将不进行深入讨论。下面的重点放在 Caffe 中的 L1 正则优化上。

8.4.1 MNIST 的 L1 正则实验

下面要进行一个展示 Caffe 中 L1 正则效果的实验。依然采用 MNIST 作为实验的数据集。训练数据和测试数据同之前的实验一致，所采用的模型也是我们之前提到的 LeNet 模型，卷积层和全连接层的非线性部分采用 ReLU 函数。唯一不同的是优化的正则化的方法从 L2 正则改为 L1 正则。模型进行 10000 轮的训练后，给出了如下的测试集精确率：

0.9766

看上去 L1 正则比 L2 正则 10000 轮训练后的精度 0.9912 要差一些。既然精度已经差了一些，那么 L1 正则应该会在别的地方有优势，例如稀疏性。下面就来看看 L1 正则优化后模型在稀疏性上的表现。观察的方法是直接画出某个小范围内参数数值的直方图。

首先是 L1 正则模型的两张图，我们只关注极小的某个区间下的参数直方图，首先是 conv1 层的参数，如图 8-23 所示。

可以看出在  $\pm 10^{-7}$  的范围下，没有参数等于 0。然后是 conv2 层，如图 8-24 所示。

在  $\pm 10^{-9}$  的范围下，只有一个参数等于 0。看完了 L1 正则的结果，下面就要看看 L2 正则训练出的参数的表现结果，首先是 conv1 层，如图 8-25 所示。

然后是 conv2 层，如图 8-26 所示。

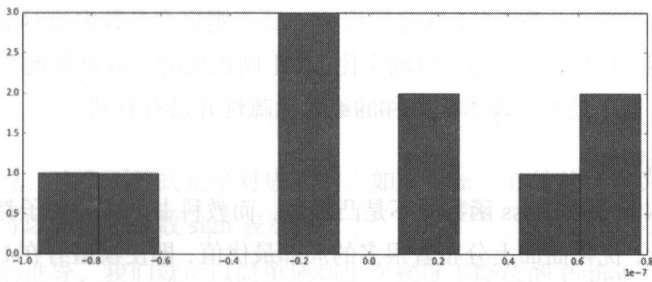


图 8-23 L1 正则下 conv1 参数的稀疏性效果展示

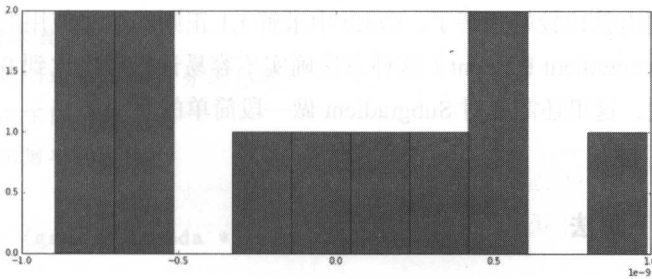


图 8-24 L1 正则下 conv2 参数的稀疏性效果展示

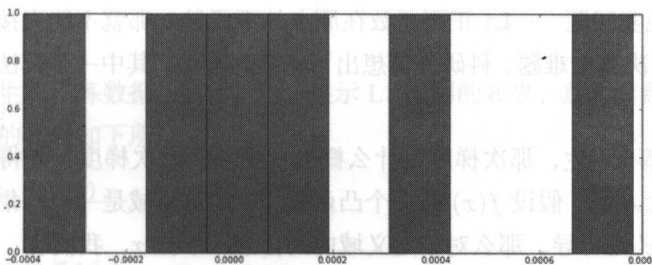


图 8-25 L2 正则下 conv1 参数的稀疏性效果展示

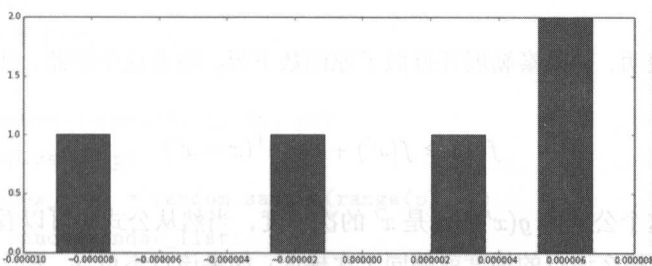


图 8-26 L2 正则下 conv2 参数的稀疏性效果展示

从分布中可以看出，和 L1 正则模型相比，L2 正则模型参数中绝对值特别小的参数比较少，这说明 L1 正则优化的参数确实比 L2 正则优化的参数更靠近 0，但可惜的是，它并没有到达 0 这个地方。课本上宣称的绝对稀疏性并没有看到。

这是为什么呢？

首先，CNN 模型的 Loss 函数并不是凸函数，而教科书上讲的例子都是凸函数，非凸函数比较复杂，优化曲面上分布着很多的局部最优值，即使模型有在 L1 正则的约束，参数的优化结果还是有可能不落在坐标轴上，所以自然有可能得不到稀疏的效果。这个问题比较复杂，很难用实际的例子讲清楚，因此这里只能从感性上理解。

另外一个理由就比较好理解了。Caffe 中求解 L1 正则的算法使用的是最基础的次梯度下降法（Subgradient Descent）这种方法确实不容易让参数优化到 0 这个数字。为了解释这个问题，这里还需要对 Subgradient 做一段简单的介绍。

#### 8.4.2 次梯度下降法

相较而言，L2 正则函数并没有什么大问题，我们确保 Loss 函数处处可导，加上 L2 正则项也是处处可导，就可以直接使用经典的梯度下降法优化模型；而使用 L1 正则会遇到一个现实问题——L1 正则函数在原点处不可导，那就不能直接使用梯度下降法优化。为了解决这个难题，科研人员想出了一系列方法，其中一个算法就是次梯度下降法。

梯度我们都不陌生，那次梯度是什么概念？要想介绍次梯度，先得从梯度和泰勒公式这两个概念入手。假设  $f(x)$  是一个凸函数， $x$  的定义域是一个凸集合，如果函数  $f(x)$  在某一点  $x'$  处可导，那么对于定义域内的任意一个值  $x$ ，我们有：

$$f(x) \geq f(x') + \nabla(f(x'))^T(x - x')$$

这个公式表明，一阶泰勒展开近似了原函数下界，顺着这个思路，就可以给出次梯度的定义：

$$f(x) \geq f(x') + g(x')^T(x - x')$$

满足上面这个公式的  $g(x')^T$  就是  $x'$  的次梯度，当然从公式中可以看出，如果函数在某一处可导，那么该点的梯度就等同于次梯度；如果该点不可导，那么该点的次梯度可能不唯一。

我们就用  $f(x) = |x|$  这个函数举例，当  $x = 0$  时，该点的次梯度可以等于 0，对于任意的  $x$ ，我们有：

$$f(x) = |x| \geq 0 + 0 \times x = 0$$

从定义来说，这个不等式是绝对成立的。如果令  $x = 0$  处的梯度为 0，那么 L1 正则项的梯度就可以用符号函数  $\text{sign}$  表示。

有了上面的推导，我们就可以简单地列出次梯度下降法的 Python 代码：

```
def subgradient_descent(x, grad, lr, lambda):
```

```
    '''
```

```
    x是当前的参数优化值
```

```
    grad是目标函数中Loss部分的梯度
```

```
    lr是次梯度下降的学习率
```

```
    lambda是正则项的权重
```

```
    '''
```

```
    x -= lr * (grad + lambda * sign(x))
```

为了验证次梯度下降算法，这里要再次进行一个实验，观察它的效果。这一次待优化的目标函数是：

$$\text{obj}(x) = \min_x \frac{1}{2}(A^T x - b)^2 + |x|$$

第一步是生成训练数据。当然，为了显示 L1 正则的效果，真实参数设计得“稀疏”些，生成数据的代码如下所示：

```
def genData(n, p, s):
```

```
    '''
```

```
    初始化训练数据A和b，
```

```
    其中A的维度为n*p，
```

```
    b的维度为n，
```

```
    参数x的维度是p*1
```

```
    '''
```

```
    A = np.random.normal(0, 1, (n, p))
```

```
    opt_x = np.zeros(p)
```

```
    random_index_list = random.sample(range(p), s)
```

```
    for i in random_index_list:
```

```
        opt_x[i] = np.random.normal(0, 10)
```

```
    e = np.random.normal(0, 1, n)
```

```
    b = np.dot(A, opt_x.T) + e.T
```

```
    return A, b
```

```
A, b = genData(100, 50, 20)
```

上面生成的优化问题中参数共有 50 维，其中有 30 个维度的参数都被随机设置为 0。最优参数 `opt_x` 的数值如下所示：

```
optx
```

```
#以下为运行结果显示
```

```
[ 0.00000000e+00  0.00000000e+00 -8.89068011e+00  0.00000000e+00
 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
 0.00000000e+00  9.79677398e+00  0.00000000e+00 -1.53016548e+01
 1.01055968e+01  2.08989507e+01  0.00000000e+00 -4.16464326e+00
 1.76916005e+01  0.00000000e+00  0.00000000e+00  0.00000000e+00
 0.00000000e+00  0.00000000e+00  0.00000000e+00  5.00900104e+00
-9.35102012e-01  0.00000000e+00  0.00000000e+00  0.00000000e+00
 0.00000000e+00 -7.13791761e+00  0.00000000e+00  0.00000000e+00
 0.00000000e+00 -2.79487453e+01  3.11610679e+00  0.00000000e+00
 0.00000000e+00  2.14217260e+00  6.00126451e+00  6.58569031e+00
 4.52257852e+00  0.00000000e+00 -2.68944021e-02  0.00000000e+00
 0.00000000e+00  0.00000000e+00 -5.51801700e+00  1.31847462e+00
-1.81286404e+01  0.00000000e+00]
```

由于目标函数中使用了 L1 正则，因此大家都希望次梯度下降算法可以学到这样具有稀疏性质的参数，具体的优化过程如下所示：

```
def obj_func(A, x, b, lmd):
```

```
    f_value = math.pow(np.linalg.norm(np.dot(A,x.T) - b),2) / 2
```

```
    g_value = lmd * np.linalg.norm(x,1)
```

```
    return f_value + g_value
```

```
def subgradient_process(A, b, lmd):
```

```
    x = np.array([0.0]*p)
```

```
    step_length = 10
```

```
    while(True):
```

```
        obj_value = obj_func(A, x_k, b, lmd)
```

```
        while(True):
```

```
            prime_grad = np.dot(A.T, np.dot(A, x_k.T) - b)
```

```
            x_new = subgradient_descent(x, prime_grad, step_length, lmd)
```

```
            obj_value2 = obj_func(A, x_new, b, lmd)
```

```

    if obj_value2 <= obj_value:
        break
    else:
        step_length = step_length * 0.5 # BackTrace
    if abs(obj_value - obj_value2) < 1e-6:
        break
    else:
        x = x_new
return x

```

上面的代码有一点复杂，优化过程包含两层循环，其中外层循环表示优化的迭代，内层循环则是 Backtrace 步长搜索方法。它通过循环尝试找寻满足优化目标的步长。

当训练结束后，参数训练结果和预想的有些偏差，虽然模型参数得到了很好的收敛，但是没有一个参数值真正等于 0，只是有大量的参数接近 0。次梯度下降法得出的结果参数如下所示：

#subgradient\_process 的运行结果显示

```

[ 1.71781432e-07 -1.47628627e-07 -8.74652150e+00 -1.72402237e-07
 5.39554174e-08 2.85199917e-07 2.09696332e-07 5.44626181e-08
 1.42748391e-07 9.09983860e+00 -1.34269176e-07 -1.48617035e+01
 9.73235939e+00 2.00436077e+01 2.01599098e-07 -4.07292753e+00
 1.69462946e+01 2.24365336e-01 -1.00920339e-04 -5.21839170e-03
 -9.68581468e-04 -1.93440885e-07 3.31274645e-01 4.36451104e+00
 -9.05097771e-01 -1.58529600e-07 9.90021826e-08 -2.60871187e-01
 1.40025608e-07 -6.60147937e+00 -7.96634585e-05 1.24702253e-07
 4.38641043e-08 -2.70520316e+01 2.73945615e+00 2.80055535e-07
 2.02949781e-07 2.52015283e+00 5.89734589e+00 6.33872176e+00
 4.64813524e+00 1.17230844e-07 -1.24874200e-01 -2.53462109e-02
 3.01514888e-07 2.72601191e-07 -5.18595641e+00 1.24439798e+00
 -1.77537445e+01 -3.96211798e-08]

```

难道是课本上的理论产生了错误？当然不是。采用其他的次梯度算法时，模型参数就可以达到稀疏的效果，例如近端梯度下降法（Proximal Gradient Descent），它的代码如下所示，由于不是本书重点，这里就不对这个方法做详细介绍了：

```

def gradient_descent(x_k, step_length, A, b):
    f_prime = np.dot(A.T, np.dot(A, x_k.T) - b) # derivitive of f_function
    y = x_k - step_length * f_prime
    return y

```

```
def prox_operation(lmd,step_length,y):
    new_y = sign(y)
    prox_vec = new_y * vec_max(0, np.absolute(y) - step_length*lmd) #
    NOTE: here is step_lengt    h * lmd
    return prox_vec

def proximal_process(A, b, lmd):
    x_k = np.array([0.0]*p)
    step_length = 10
    while(1):
        obj_value = obj_func(A,x_k,b,lmd)
        while(1):
            grad = gradient_descent(x_k, step_length, A, b)
            x_k_plus_1 = prox_operation(lmd,step_length, grad)
            f_value = f_func(A,x_k_plus_1,b)
            m_value = m_func(A,x_k, b, step_length, x_k_plus_1)
            if f_value <= m_value:
                break
            step_length = step_length * 0.5
        if f_func(A,x_k,b) <= f_func(A,x_k_plus_1,b):
            break
        else:
            x_k = x_k_plus_1
    return x_k
```

这个方法得到的最终参数如下所示：

#proximal\_gradient\_process的结果运行显示

```
[ -0.          0.          -8.65370345   0.
  0.          0.          0.          0.
  0.          9.30621127  -0.         -15.07056666
  9.76129302  20.45180052  0.         -4.11430611
 17.16825959  0.         -0.         -0.
 -0.         -0.          0.          4.55458453
 -0.81317833  0.         -0.         -0.07547525
 -0.         -6.73065609  -0.         -0.
 -0.         -27.54904074  2.87046799  0.]
```

```

0.          2.23054671   5.8947849   6.36406299
4.61893832 -0.          -0.          -0.
0.          0.          -5.26100017   1.19033131
-17.77335708 -0.          ]

```

这个方法的稀疏性效果一目了然，在把参数截断成 0 这个方面，近端梯度下降法比次梯度下降法要好很多。

综上所述，在 Caffe 中利用 L1 正则获得参数的稀疏性是不太可能的。当然，还有一个更重要的问题，L1 的稀疏性真的是我们需要的吗？答案是否定的。这个问题还是要从稀疏性的源头说起。参数稀疏能带来什么好处？这主要从参数存储和模型计算两个方面考虑。

参数稀疏化后模型需要存储的参数变少了，因此如果稀疏性足够强，采用稀疏矩阵的方式存储，模型需要的空间会变小；如果用 CPU 进行运算，由于有大量的参数不需要计算，利用一些稀疏矩阵的计算方式，模型的计算会变快。如果采用 GPU 计算，稀疏性并不能减少模型计算的时间。

那么参数的存储空间能省多少呢？这个问题实际上涉及了另外一个大问题——模型压缩，这个问题不是本书的重点，欢迎感兴趣的读者查阅相关资料深入了解。

## 8.5 寻找模型的弱点

本节的标题可能很吸引眼球，但是它的原理实际上很普通。现如今大家看到了 CNN 模型的强大，强大到很多曾经称霸机器学习界的模型都黯然失色。但大家还是忍不住想到一个问题：CNN 是否已经足够完美？当然不够。说到底，CNN 只是一个高维度的非线性函数，它逃不过所有机器学习模型需要面对的问题——泛化问题。例如，一个用于人脸识别的 CNN 模型，这个模型在训练数据集和测试数据集上表现良好，但是在未来的使用过程中还是有可能遇到一些被误判的用例。那么问题来了，能不能提前找出这些问题数据，然后想办法把这些漏洞补上呢？

这里读者可以想象下面两个场景。

1. 如果模型的泛化性足够好，那么即使对之前识别正确的数据做轻微的改动，模型还是有可能识别正确的；但是如果没有识别正确，这些 False Positive 的错误用例（bad case）就可以证明模型的泛化性还有待提升。举个极端的人脸识别的例子，如果因为一个人脸上长了一颗痘痘而将她识别为另外一个人，那就说明模型的泛化性还存在缺陷。



2. 如果模型的精确度足够好，对于一些本来不应该识别出类别的图像，模型就不应该识别出来，或者说识别的信心度比较低；若竟然识别出来了，而且识别得十分正确，那么这些 False Negative 的错误用例就证明模型的精度还有待提升。举个极端的例子，同样是人脸识别，如果送来识别的图像是狗，而识别竟然通过，那就说明模型的精确性还存在缺陷。

针对上面的问题，本节将设计两个实验来寻找 CNN 的错误用例。实验同样将以 MNIST 数据集为例，所使用的模型也还是之前的 LeNet 模型结构，如图 8-27 所示。

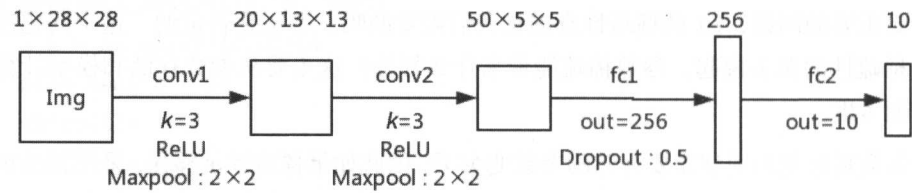


图 8-27 LeNet 模型的网络架构

两个实验的流程大体一致，如下所示。

1. 将每张图像做一点轻微的改动，确保这个改动对人来说不会造成识别结果的改变。
2. 把改动后的图像传入 CNN 模型做预测，看看 CNN 的预测结果和之前相比有没有发生改变。
3. 根据结果判定实验是否停止，如果不停止，则返回步骤 1。

8.5.1 泛化性实验

第一个实验要观察在 CNN 模型保持识别结果不变的情况下，输入的图像可以变成什么样。也就是说，只要识别结果和最初的识别结果相同，实验将不会停止。

对于一个数据范围限定在 0~1 的训练数据，我们在每一轮迭代中会生成 64 个随机变化的图像，每个图像的每一个像素点会随机加上-0.1~0.1 的一个数。经过 CNN 模型预测，就可以得到这 64 个图像的预测 label。接下来，从 64 张图像中选择一张和原始图像 label 相同的图像保存起来，作为下一轮迭代操作的图像。然后进行下一轮迭代……经过若干轮迭代后，程序将生成的图像输出，就可以看到这个被随机改变的图像变成了什么样子，也可以看看对于同一个 label，模型能够接受变化的范围有多大。

首先选择一个数字 0，如图 8-28 所示。

经过 50 轮迭代，程序给出了如图 8-29 所示的图像，还是可以依稀辨别出 0 的样子。

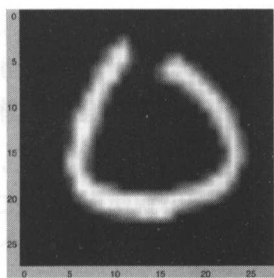


图 8-28 实验数据的原图

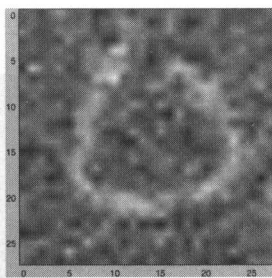


图 8-29 50 轮随机变换后的图像

经过 100 轮迭代，程序得到了如图 8-30 所示的图像，这时的 0 已经不太容易辨别了。

经过 150 轮迭代，程序得到了如图 8-31 所示的图像，此时已经很难看清数字了。

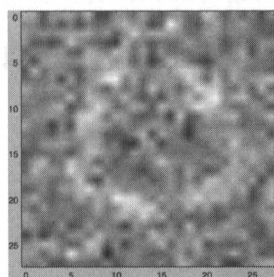


图 8-30 100 轮随机变换后的图像

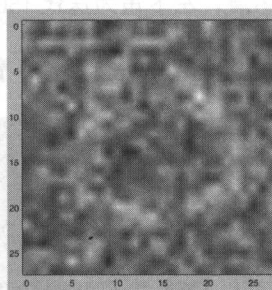


图 8-31 150 轮随机变换后的图像

经过 200 轮迭代，程序得到了如图 8-32 所示的图像，这时的图像和噪声图像已经差距不大了。

到此为止，图像已经变得模糊不清，大量杂乱的信息混入其中，已经和原始的数字完全不同。除非从前面的图片一步一步看起来，否则想辨别出这些数字实在太难。

那么，换一个数字后的结果如何呢？再看看另一个数字 5，如图 8-33 所示。

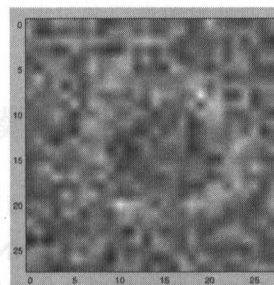


图 8-32 200 轮随机变换后的图像

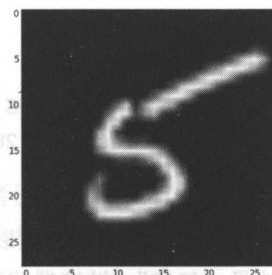


图 8-33 数字 5 的原图

经过上面的变换，图像变成了如图 8-34 所示的样子。

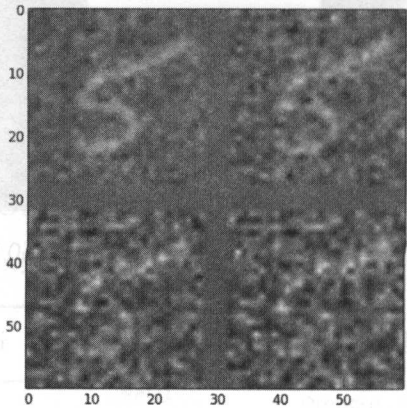


图 8-34 随机变换后的数字图像，左上角为迭代 50 轮的图像，右上角为迭代 100 轮的图像，左下角为迭代 150 轮的图像，右下角为迭代 200 轮的图像

如果用其他数字进行变换，依然可以得到类似的结果。上面的实验可以用下面这段代码实现：

```
import matplotlib.pyplot as plt
def gen(img, label, model):
    for iterator in range(200):
        fake_imgs = image_generator(img, 64)
        pred = model.predict(fake_imgs, batch_size = 64)
        pred_label = np.argmax(pred, axis=1)
        flag = False
        for i in range(64):
            if pred_label[i] == label:
                choosed_img = fake_imgs[i]
                flag = True
                break
        if flag == False:
            break
    else:
        img = choosed_img
plt.imshow(img.reshape((28,28)), cmap=plt.cm.gray)
plt.show()
```

代码内容本身并不复杂，所以不再详细介绍。这个套路被称作“fool CNN”<sup>[7]</sup>，用东北话说就是“忽悠”。继续迭代下去，程序还能生成更多令人不可思议的图像。当然，

这也只是忽悠 CNN 模型的一种办法，实际上还有其他办法生成图像。这种“忽悠”的方法其实也可以理解，因为模型只有 10 个类别，而图像的空间如此之大，无论什么样的图像，模型总会给它一个主要类别，那么这些非数字图像就会混入其中。

另外，上面给出 CNN 的模型说到底还是个判别式模型，如果把图像设为  $X$ ，label 设为  $y$ ，CNN 的模型就相当于直接求  $p(y|X)$  的值。判别式模型相当于描述“什么样的图像是这个 label 的图像”，而满足了这些条件的图像有时并不是具有真实 label 的那个图像。上面的实验就利用了这个漏洞。

### 8.5.2 精确性实验

上面的实验用 fool CNN 的方法让一张模糊不清的图像保持了原来的 label，当然，同时也可以让一张不算模糊的图像被 CNN 错认成另外一个 label。在第二个实验中，只要发现图像的识别 label 和最初的 label 不同，实验就停止下来，并保存相应的图像。

例如，下面这张经过 40 轮迭代的图像被错认成了 6，如图 8-35 所示。

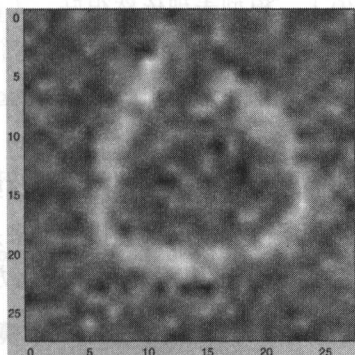


图 8-35 精确性实验中错认的图像

这些套路的出现让人对 CNN 有了一些警惕，如果想让 CNN 对手写数字的识别完全精确，补充数据是一件不能停止的事情，不然这种意外总会发生。除此之外，从模型结构入手解决这样的事情也是十分必要的。

## 8.6 模型优化路径的可视化

前面章节介绍了很多优化算法的内容，虽然读者已经对优化有了一定的了解，但是对于一个复杂的深层模型的优化问题来说，这些内容还不够直观。可不可以有更直观的方式让大家深入模型优化的内部呢？答案是可以的。

*Qualitatively Characterizing Neural Network Optimization Problems*<sup>[8]</sup> 这篇文章从一个十分有趣的角度为大家展示了这些复杂的优化问题背后简单的一面。大家都觉得深层模型的参数难以优化，但是这篇文章的实验结果却告诉我们，在某些问题中，找到局部最优并不是一件困难的事情。

已经有很多研究人员通过各种方式证明了一个事情——随着模型的层数和参数数量的增加，模型参数的优化曲面变得十分复杂，但这对获得高质量的模型并没有造成很大的困难。对于经典的优化理论来说，这个现象并不是很友好，因为随着模型变得复杂，模型的局部最优点变多，想要找到全局最优点变得十分困难。不光是全局最优点很难找到，优化曲面中还存在一些鞍点（Saddle Point），或者一些梯度很小的“平原带”。一旦参数值进入了这些区域，模型的目标 Loss 可能会停止不动，这时的我们很难判断模型的状态——是已经接近局部最优点，即将完成收敛，还是进入了平原，没有了梯度？这让训练的难度增大了不少。

但幸运的是，科研人员也利用各种抽象的模型证明了一点：随着模型复杂度上升，局部最优点的目标函数值和全局最优点的函数值的差距在不断缩小<sup>[11][12]</sup>。这样即使模型参数收敛到一个局部最优值上，模型表现依然很好。不必像经典的凸优化理论那样追求全局最优，这对于奋战在模型训练第一线的广大同胞来说简直就是一个福音。

而本节将要介绍的这篇论文又带来了另外一个福音——实际上从某一初始点到局部最优点的路也不总是崎岖难行的。这一点该如何证明呢？在本章前面的小节中，由于我们精心挑选了一些低维度的问题，它们的优化曲面可以轻松地在图上，但是对于参数达到几十万甚至上百万的深层模型来说，直接画出来是基本不可能的。为了解决这个问题，论文的作者带来了一种特殊的优化曲面可视化方法。

想要理解这个问题，只能从某个切面入手。如果一个模型已经训练完成，那么优化曲面上就有了两个有意义的点——一个是优化的起始点，一个是优化的终点。大家都知道两点可以连成一条线段，也知道两点之间线段最短。那么一个最直观的想法就出现了：能不能把从优化起始点到终点这条路径上的目标函数值画出来，通过观察这条路径，至少可以帮助我们回答一个问题：这条线段上的道路好不好“走”，优化算法有没有可能是走了这条线段到达了终点呢？

从这个角度想，这个问题就变得有意思了许多。如果深层模型真的难以训练，优化曲面上到处都是陷阱，到处是不够优秀的局部最优点和鞍点，那么初始点和终点之间一定充满了障碍；如果深层模型并没有难以训练，那么也许这两点之间一片坦途，即使十分简单的优化算法也可以完成优化。

思路有了，下面就该看看这个问题该如何实现了。本节将使用另一种开源框架——TensorFlow 进行演示，演示使用的数据集同样是 MNIST，这个数据集比较简单，当然

得到的结果也更符合这篇文章给出的结论。

首先给出 LeNet 模型的 TensorFlow 代码。代码总体比较直观，即使没有学过的读者也基本可以看懂里面的内容：

```
class MNist:
    def __init__(self):
        self.X = tf.placeholder('float', [None, 28, 28, 1])
        self.Y = tf.placeholder('float', [None, 10])
        predictY = self.model(self.X)
        self.cost_op = tf.reduce_mean(tf.nn.
            softmax_cross_entropy_with_logits(predictY, self.Y))
        self.predict_op = tf.argmax(predictY, 1)
    def model(self, X):
        w = tf.Variable(tf.random_normal([5, 5, 1, 20], stddev=0.01))
        w2 = tf.Variable(tf.random_normal([5, 5, 20, 50], stddev=0.01))
        w3 = tf.Variable(tf.random_normal([4*4*50, 500], stddev=0.01))
        w4 = tf.Variable(tf.random_normal([500, 10], stddev=0.01))
        l1 = tf.nn.relu(tf.nn.conv2d(X, w, strides=[1,1,1,1], padding='
            VALID'))
        l1p = tf.nn.max_pool(l1, ksize=[1,2,2,1], strides=[1,2,2,1],
            padding='VALID')
        l2 = tf.nn.relu(tf.nn.conv2d(l1p, w2, strides=[1,1,1,1], padding='
            VALID'))
        l2p = tf.nn.max_pool(l2, ksize=[1,2,2,1], strides=[1,2,2,1],
            padding='VALID')
        l2p = tf.reshape(l2p, [-1, 4*4*50])
        l3 = tf.nn.relu(tf.matmul(l2p, w3))
        y = tf.matmul(l3, w4)
        self.params = [w, w2, w3, w4]
        return y
```

由于训练过程和这个实验无关，因此这里跳过具体的训练步骤，只列出保存模型的部分代码：

```
class MNist
    def train(self, trainX, trainY, testX, testY):
        train_op = tf.train.RMSPropOptimizer(0.001, 0.9).minimize(self.
            cost_op)
        with tf.Session() as sess:
```

```

saver = tf.train.Saver()
tf.initialize_all_variables().run()
saver.save(sess, 'init.ckpt')
# 以下是训练过程
saver.save(sess, 'model.ckpt')

```

经过训练，模型的初始参数和优化完成的参数都被保存在 checkpoint 文件中，下面的操作就要针对这两组参数进行了。首先要做的是把参数从 checkpoint 文件中读取出来：

```

model = MNist()
with tf.Session() as sess:
    saver = tf.train.Saver()
    saver.restore(sess, 'init.ckpt')
    init_weights = [sess.run(param) for param in model.param_list]
    saver.restore(sess, 'model.ckpt')
    opt_weights = [sess.run(param) for param in model.param_list]

```

获得了两组参数后，下面就要计算出两点之间的参数状态，同时遍历所有的测试数据，求出测试结果：

```

with tf.Session() as sess:
    tf.initialize_all_variables().run()
    for rate in range(0, 101):
        for i, param in enumerate(model.param_list):
            temp_val = (init_weights[i] * (100 - rate) + opt_weights[i] *
                        rate) / 100
            assign_op = param.assign(temp_val)
            sess.run(assign_op)

        acc = 0
        num = 0
        length = len(data)
        batch_size = 128
        test_batch = zip(range(0, length, batch_size), range(batch_size,
                                                                length + 1, batch_size))
        for start, end in test_batch:
            loss = sess.run(model.cost_op, feed_dict={model.X: data[start:
                                                                end], model.Y: label[start:end]})
            acc += loss * (end - start)

```

```

num += end - start
print rate, acc / num

```

上面的代码选取了从初始点到终点的 101 个中间状态，并求出这些中间状态模型的精度。以下是这条路径上的目标函数形成的函数图像，如图 8-36 所示。

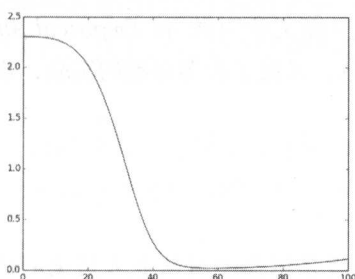


图 8-36 从初始点到优化终点的目标函数曲线

图中的  $x$  轴表示参数路径上的位置，0 表示测试参数完全由初始化参数组成，100 表示测试参数完全由最优参数组成， $y$  轴为 Loss 值。从结果可以看出，初始化点和终点之间确实一片坦途，优化算法只要一路向前前进即可到达目标，完全不需要任何套路。这可能和想象中那个百转千回到达目标的画面不太一样。这个实验似乎像读者展示了深层模型优化的一个演化路线。

1. 模型复杂度：随着参数数量增大和模型深度加深，获得模型的全局最优值变得越来越难。
2. 模型优化满意度：随着模型变得越来越复杂，局部最优点的竞争力不断提高，以至于找到一个附近的局部最优值就可以达到满意的效果。

那么为什么附近的局部最优值可以令人满意呢？这也来自于优化曲面的非凸性。大家都知道凸函数的性质——局部最优等于全局最优。而非凸问题，尤其是深层神经网络模型，很多不同的局部最优解相当于全局最优解的不同表达形式。

就拿 MNIST 数据集来说，虽然它处于 764 维的空间中，但实际上这些数字图像所组成的真实数据分布很可能不足 764 维；而识别数字的模型的参数维度是大于 764 维的。如果把神经网络抽象成一个线性变换的过程，那么这个问题就相当于求解一个参数数量多于方程数量的方程组。这样的问题是很有可能有多于一个解的。如果模型进一步变得复杂，就相当于参数数量进一步增加，那个解的数量也会变得更多，所以在深层神经网络中存在参数不同但结果相近的现象是很有可能。

基于这样的结论，大家就可以更加宽容地面对自己得到的“局部最优解”了，因为从任何角度来看，它真的和全局最优解差不多。



当然，这条曲线还暴露了训练过程中的另外一个问题。可以看出  $x > 50$  后，模型在测试集上的 Loss 开始上升，从某种意义上说，模型产生了“过拟合”的现象。那么  $x = 50$  的这个模型是否曾出现在优化过程中呢？如果出现过，那么将进一步证实模型“过拟合”的现象，我们应该减少模型训练的轮数；如果没有出现过，说明模型并没有走这条直线，那么模型优化的路径就变得更奇妙了。

除了这篇文章，另外有一篇文章叫做 *An Empirical Analysis of Deep Network Loss Surface*<sup>[9]</sup>，它基于上面的文章，又做了很多有趣的实验，欢迎各位读者自行阅读。

## 8.7 模型的过拟合

模型的泛化性一直以来就是机器学习从业人员关心的大话题。一篇名为 *Understanding deep learning requires rethinking generalization*<sup>[10]</sup> 的文章抛出了一个有关深层模型泛化问题的观点，引起了大家的广泛关注，而且在各种社交媒体上引发了一系列的讨论。下面就来看看文章给出的论断。

论文首先提出了一个看上去并不新鲜的论断：深层模型是可以“背下所有训练数据的”。为了证明这一点，作者先用一个“深到没朋友”的复杂模型去解决一个相对简单的 CIFAR10 数据集问题（说实话现在很多模型都可以做到 100% 的训练集精度），接下来就开始了一系列充满想象力的实验。

1. 篡改数据的 label，将训练数据里的 label 作随机变换。
2. 篡改图片，将训练数据里的图像像素做改变。

经过这两种改变，图像信息和类别信息在语义上已经无法匹配了，如果让一个正常人来看的话，是无法区分这些图像对应的类别的。那么对于深层模型来说，这样的任务可以学习吗？文章通过实验给出的结论是——当然可以。即使问题变得毫无逻辑，在冷酷的深层模型面前，这个问题一样可以解决。

面对这样的“超能力”，相信读者也会觉得——模型会不会把题目都背下来了？这个现象可以从两个方面理解。

1. 站在人类常识的角度去理解，模型确实有可能背下来了。但是如果抛开人类几千年的文明来看，也许存在某一种奇异的生物，它恰好可以解码某种特殊的图像——标签对呢？那些随机图片在它们的文明世界有某种具体的代表呢？这种感觉和我们见到一种不认识的语言类似。如果我们认同这些图像—标签对背后可能存在的逻辑和文化，那么模型的表现就没那么奇怪了。
2. 站在人类对于某些复杂问题处理方式的角度去思考，这个现象也并非难以理解。相信读者也曾见过或者亲身经历过这样的场面：当一位同学在一门考试前紧急备

考时,面对各种各样复杂的问题,这位同学的大脑已经无法处理这些抽象到难以理解的问题,这时他通常都有一个保底方案——记住答案,强行让问题与答案建立联系。在这一刻,即使问题和答案实际上是具有逻辑并且是可以解释的,对于同学的大脑来说,它就是一些难以理解的符号和数字,这一刻他凝视这些题目的感觉,就好像我们在看那些古老的已经失传的文字一样。所以“背下来”也是人类解决这类问题的一种手段,只不过人类的背和机器的背还不太一样,这里只是打个比方。

当然,从实验本身来看,过拟合的现象是一定存在的。为了证明神经网络具有过拟合的能力,作者给出了一个实现网络过拟合的可行性方案。下面就来详细介绍这个方案。

### 8.7.1 过拟合方案

我们假设有一个简单的数据集:

```
x^T=[
    [0.2, 0.3, 0.6],
    [0.5, 0.7, 0.1],
    [0.3, 0.4, 1],
    [0.3, 0.3, 0.3]
]
```

它的 label 如下所示:

```
y=[1 1 0 1]
```

我们能不能构建一个网络,使得这个数据集完美拟合呢?答案是可以的。对于一个维度为  $d$  的  $n$  个数据组成的数据集,我们可以用  $2n + d$  个数据将数据集完美拟合。对于这个问题,只需要一个有 11 个参数的 2 层神经网络。其中第一层的输出为  $n$ ,也就是 4。首先选择一个第一层的参数:

```
w1=[
    [1 1 1]
    [1 1 1]
    [1 1 1]
    [1 1 1]
]
```

这个参数有一个特点，那就是每一行的参数都一样，下面几行的参数实际上是复制了第一行的参数。这样经过点乘，可以得到：

```
w1*x=[
    [1.1 1.3 1.7 0.9]
    [1.1 1.3 1.7 0.9]
    [1.1 1.3 1.7 0.9]
    [1.1 1.3 1.7 0.9]
]
```

其实前面的章节中曾介绍过，这种参数完全一致的模型实际上是一种退化了的模型。下面的过程就有技巧了，我们设计了一些特殊的偏置项：

```
b1^T=[-0.8, -1.0, -1.2, -1.6]
```

计算后得到：

```
w1*x+b1=[
    [0.3, 0.5, 0.9, 0.1]
    [0.1, 0.3, 0.7, -0.1]
    [-0.1, 0.1, 0.5, -0.3]
    [-0.5, -0.3, 0.1, -0.7]
]
```

然后让这个网络经过 ReLU，得到：

```
ReLU(w1*x+b1)=[
    [0.3, 0.5, 0.9, 0.1]
    [0.1, 0.3, 0.7, 0]
    [0, 0.1, 0.5, 0]
    [0, 0, 0.1, 0]
]
```

我们用这个中间结果乘以某一个参数  $w_2$ ，让它得到最终结果  $y$ 。采用反解的方式可以把  $w_2$  求解出来：

```
w2^T = [
    0.9, 0.4, 0.1, 0
]
```

通过上面的推演问题得到了解决，完美拟合成为了可能。下面让我们用形式化的方法介绍 2 层神经网络完美拟合任意数据的方案。

这里  $\mathbf{x}$  是  $d$  维的,  $\mathbf{w}^1$  是  $n \times d$  维的, 但是由于每一行的数据是相同的, 所以它相当于只有  $d$  维参数。 $\mathbf{b}^1$  是  $n$  维的,  $\mathbf{w}^2$  是  $n$  维的。这就是我们所使用到的所有参数:  $2n + d$ 。

下面的第一步, 我们希望找到一组参数  $\mathbf{w}^1$ , 使得不同的数据与它相乘的结果是不同的。这里用  $z$  来表示, 也就是说对于  $n$  个数据, 存在某种排序  $pn$ , 使得:  $z_{p1} < z_{p2} < z_{p3} < \dots < z_{pn}$ 。我们不妨假设  $p1 = 1, p2 = 2$ , 于是, 这个  $n \times n$  的矩阵就变成了:

```
z = [
[z1, z2, ..., zn]
[z1, z2, ..., zn]
.....
[z1, z2, ..., zn]
[z1, z2, ..., zn]
]
```

第二步, 由于这个序列是一个有限序列, 于是就可以找到另一个序列  $\{b_n\}$  用于表示偏置项, 这个序列有下面的特点:

$$b_1 < z_{p1} < b_2 < z_{p2} < \dots < b_n < z_{pn}$$

这样经过计算, 再经过 ReLU 层, 就得到了:

```
[
[z1-b1, z2-b1, ..., zn-b1]
[0, z2-b2, ..., zn-b2]
[0, 0, z3-b3, ..., zn-b3]
...
[0, 0, 0, ..., 0, zn-bn]
]
```

可以看出这是一个上三角矩阵 (这里和原论文的证明稍有不同, 论文中得到的是下三角矩阵), 只要对角线不为 0, 这个矩阵就是一个满秩的矩阵, 对于形如  $\mathbf{Ax} = \mathbf{b}$  的解方程问题, 方程存在唯一解, 因此, 无论最终的标签结何, 从上面的矩阵出发都可以得到正确的结果。

## 8.7.2 SGD 与过拟合

论文中还提到关于 SGD 对过拟合的防止作用, 其实从深层模型的角度, 还可以有别的理解方法。前面的章节中曾介绍过参数的稳定性问题。为了保证参数稳定, 参数的初始化及更新过程都会保证每一层参数的均值方差控制在一定的范围内, 这就相当于

对参数进行限制。如果深层模型的参数像浅层模型那样可以野蛮生长，那么带来的结果并不仅仅是过拟合，更有可能的是直接溢出。

所以从这个角度看，为了保证模型是可用的，优化过程已经隐含了 L2 正则这样的正则项。所以我们再加入 L2 正则项虽然可以起到效果，但确实不是唯一起作用的。

### 8.7.3 对于深层模型泛化的猜想

既然论文中提到了深层模型的过拟合能力和本身所具备的泛化能力，那么这个泛化能力究竟是从何而来呢？作者在此提一点自己的猜想。作者认为通过过拟合来实现模型泛化也是有希望的，但是需要基于一个前提：模型的输入空间——也就是图像空间在某个邻域内具备泛化能力。

可以想象一下传统的机器学习模型，它们的模型结构相对清晰好分析，于是前辈们可以用一些量化指标分析模型的复杂程度，比方说像 VC 维那样的指标。但是到了深层模型这里，模型表示和优化都发生了变化，很多曾经的指标变得不那么好用了。在机器学习的课本上曾经讲过，2 阶模型的泛化性比 10 阶模型的泛化性强，但是对于深层模型来说，参数数量、模型结构对模型的复杂度都会产生影响，大家都是 10 阶复杂度的模型，所以从模型结构判断复杂度已经不太容易了。

那么深层模型的泛化从何而来？由于深层模型毕竟还是一个可导的模型，我们就不由地猜想，这个模型是否满足 Lipchitz 条件这样的性质，也就是说模型的函数值波动的范围有限？对于某一个训练数据，如果模型已经将它拟合得足够好，那么：

1. 对训练数据稍作改动，但不改变数据的本质，这个映射是否还能保持正确呢？
2. 对训练数据作关键性改动，这个映射的结果是否会改变呢？

8.5 节已经谈过类似的话题，结果大概可以佐证，我们训练的模型是可以满足这个条件的，只不过满足条件的邻域不会很大。而且这个邻域绝不是我们平时想象的开球集合：

$$B_{\epsilon}(x) = \{y | d(x, y) < r\}$$

它应该是一个比较奇怪的形状。

如果这个条件说得通，那么深层模型的泛化问题就可以用这样的方式描述了。

1. 我们知道两个集合  $\mathbf{X}$  和  $\mathbf{Y}$ ，存在某个映射使得每一个  $x$  都对应一个  $y$ 。
2. 有训练数据对  $\{x, y\}$ ，其中  $x \in \mathbf{X}, y \in \mathbf{Y}$ ，经过深层模型计算，这些点的映射都可以被完美地学习出来。
3. 根据上面的假设， $x$  附近的邻域由于模型的局部泛化能力而得以满足。

4. 如果训练数据足够多，邻域的范围足够大，以至于它们组合起来可以将整个输入集合“盖住”或者“基本盖住”，就可以认为模型通过拟合的手段获得了泛化能力。

按照上面的猜测，对于深层模型来说，模型结构自身复杂度带来泛化能力的影响可以忽略不计，深层模型的 VC 维都非常高。L2 正则这样的约束属于模型自身必备的属性，也不必单独提出来，那么留给模型增强泛化能力的手段还有两类。

1. 补充足够多的数据——这是每一个做深度学习应用的人都在做的事情。
2. 让局部泛化的范围变大——这就有些抽象了。L2 正则限制参数的大小，可以让模型在局部表现得比较平滑，从而起到一些效果；Batch Normalization 也限制了模型，所以某些时候它可以起到一些效果。

未来会有更多的专家从第 2 点发力，发明更多让模型更容易满足局部泛化性质的结构，这样模型整体的泛化能力就更容易提高了。

## 8.8 总结

本章主要介绍了优化与训练的一些内容，让我们来回顾一下。

- 梯度下降法的步长和待优化的函数有很强关系。
- 动量算法能够帮助目标点“穿过优化曲面中的山谷”。
- 动量和梯度自适应是目前一阶梯度优化方法的两个主要方向，每个算法都有自己的长度和弱点。
- 深度学习中想通过优化 L1 正则使参数满足稀疏性是一件困难的事情。
- 模型训练不光有常规数据的训练，有时也需要一些对抗数据的训练。
- 观察参数的优化路径是了解模型优化过程的一种很好的方法。
- 在深层模型中，模型的结构复杂度和浅层模型的度量方法不同，模型过拟合的意义也变得不大一样。

## 8.9 参考文献

[1] Sutskever I, Martens J, Dahl G, et al. On the importance of initialization and momentum in deep learning[C]// International Conference on Machine Learning. 2013.

[2] A method for unconstrained convex minimization problem with the rate of convergence[C]//  $O(1/k^2)$ . Soviet Mathematics Doklady. 1983.

[3] Duchi J, Hazan E, Singer Y. Adaptive Subgradient Methods for Online Learning and Stochastic Optimization[J]. Journal of Machine Learning Research, 2011, 12(7):2121-2159.

[4] T. Tieleman, and G. Hinton. RMSProp: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural Networks for Machine Learning. Technical report*, 2012.

[5] Zeiler M D. ADADELTA: An Adaptive Learning Rate Method[J]. Computer Science, 2012.

[6] Kingma D P, Ba J. Adam: A Method for Stochastic Optimization[J]. Computer Science, 2014.

[7] Nguyen A, Yosinski J, Clune J. Deep neural networks are easily fooled: High confidence predictions for unrecognizable images[J]. 2015:427-436.

[8] Goodfellow I J, Vinyals O, Saxe A M. Qualitatively characterizing neural network optimization problems[J]. Computer Science, 2014.

[9] Im D J, Tao M, Branson K. An Empirical Analysis of Deep Network Loss Surfaces[J]. 2016.

[10] Zhang C, Bengio S, Hardt M, et al. Understanding deep learning requires rethinking generalization[J]. 2016.

[11] Choromanska A, Henaff M, Mathieu M, et al. The Loss Surfaces of Multilayer Networks[J]. Eprint Arxiv, 2015:192-204.

[12] Kawaguchi K. Deep Learning without Poor Local Minima[J]. 2016.

# 9

## 应用：图像的语意分割

前面章节中大量的例子都围绕图像分类问题展开，本章主要关注图像领域的另一个问题：图像分割的问题。图像分割也是视觉领域一个十分重要的问题，在深度学习来临之前，科研人员曾创造了许多图像分割的方法，它们基于各种各样的原理和规律，本章要介绍深度学习模型在解决这个问题上的一些方法。

首先，我们来简单看看图像分割的任务是什么样的。所谓的图像分割就是确定一幅图像中各个实体的边界，这样我们可以通过寻找这个实体的边界确定实体所在的位置。在自然场景中，程序可以根据画面中实体所在的位置进行划分，如图 9-1 所示。在文档图片中，我们可以找出每一个文字的边界把文字和背景分离出来。这些都可以算是图像分割的任务。



图 9-1 图像分割任务示例（图片来源：Jifeng Dai, Kaiming He, Jian Sun, BoxSup: Exploiting Bounding Boxes to Supervise Convolutional Networks for Semantic Segmentation, arXiv:1503.01640.）

当然，上面的描述还不太具有可操作性，这里可以把问题做进一步转化。由于之前 CNN 模型已经解决了图像分类的问题，那么能否将分类问题模型转换成一个分割问题



模型，模型从判定整张图像的类别，变成判定每一个像素点的类别，这样判别后再将相邻且类别相同的像素点聚成一类，这样类与类的相交处不就是实体的边界了吗？这就是从分类问题出发解决分割问题的方式。

但是这种解决分割问题的方法还存在一些问题：之前对于一张图像，模型最终只输出一个结果或者几个结果，结果的数量并不多；而现在输出的数量和像素的数量相同，那么模型还能够获得和分类问题近似的精确率吗？这里存在着一些需要解决的问题，例如在分类问题中，为了增强图像 transform invariance 的能力，模型会加入 Pooling 层减小长宽维度，这个被减小的维度如何能够被再次恢复放大呢？

同时，模型也没有考虑识别类型之间的关系，如果一个像素被判定为“人”的类别，那么附近一定还有很多像素也应该被判定为“人”的类别。这样的约束 CNN 能否利用呢？这些都是本章要解决的问题。

## 9.1 FCN

**FCN**（Fully Convolutional Network）是一个不包含全连接层的网络。这里所谓的不包含全连接层，实际上并不是标榜自己没有全连接层，而是为了保证计算过程中数据的维度不发生变化。一般来说，全连接层需要把本来立体的三维图像（通道  $\times$  长  $\times$  宽）变成一维数据，这样使得原本存在的空间特性被抹掉。为了确保我们识别出来的类别能和原来每一个像素点的位置对应得上，不能粗暴地把数据变“平”，这也是网络中不使用全连接层的原因。

既然我们只使用卷积层，那么网络结构该如何设计呢？前面说到模型要求出每一个像素所属的类别，那么一个最直观的想法就是使用长宽维度不做变化的卷积核进行卷积运算，计算像素附近的相关特征。这样每次卷积操作时图像需要做一定的填充（padding），填充的大小和卷积核的大小有关。这种方案肯定是没有问题的，但是它的计算量也一定会非常大。一张需要做图像分割的图像往往包含很多图像实体，那么这张图像的大小一般会很大，在这样的图像上做卷积操作，且维度不做变化，其中的计算量可想而知。

既然保持维度不变这件事情不太靠谱，那就需要换一个思路。在图像中一个实体往往需要很多像素聚合在一起表达出来，同一块区域的像素往往表示为同一个实体，那么当卷积神经网络模型把图像的局部信息汇总到一定程度时，可不可以将图像的维度缩小一些，用汇总信息代表原始像素区域的信息呢？当然是可以的，模型可以在完成特征抽取后进行降维工作，例如加入 Pooling 层。这样代表了附近像素信息汇总的特征将接受模型的判断，得到结果后，模型再将维度扩展到图像的原始尺寸大小，给每一个像素赋予它应当属于的类别。以上思想可以用图 9-2 表示。

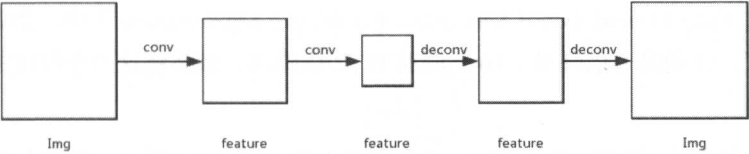


图 9-2 全像素识别的快速方案图

将特征维度扩展回图像原始尺寸的事情可以有很多解决方法：可以使用差值算法将小尺度的图像扩展到大尺度，例如使用 Bilinear Interpolation 这样的算法。当然，这种算法只是简单地把图像维度扩大，无法仔细分析维度扩展时边缘位置确定这样的比较细节的问题。如果想让图像的边缘像素计算得更精细，模型就要用到反向卷积层了。

反向卷积层并不是一个十分神秘的网络层，它实际上是将卷积层的前向后向计算颠倒过来。它的前向计算和卷积层的后向计算相同，于是如果一个卷积层的前向计算是让维度缩小，那么卷积层的后向计算就会让计算输出维度变大，同理，反向卷积层就会让维度变大。反向卷积层的结构如图 9-3 所示。

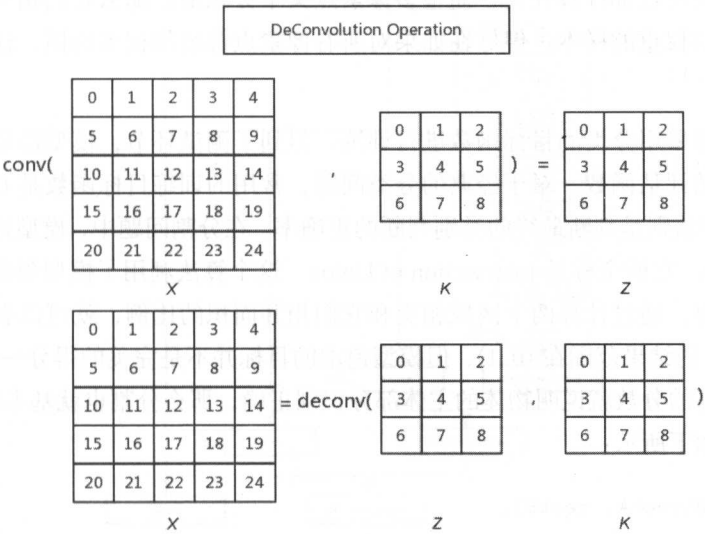


图 9-3 反向卷积操作示意图

先降维再升维的方案从技术上是可行的，那么图像该降低多少维度呢？如果想加快计算速度，降维的幅度自然要加大些；如果想确保计算的精度，降维的幅度就要尽可能的小。所以确定降维幅度也是一个需要取舍的问题。全卷积网络采用了多种降维幅度结合的方式，模型将同时保存多种尺度的特征信息，然后将这些特征信息融合起来。这样就可以在尽可能地保证计算精度的同时，保证运算速度不会太慢。

在论文 *Fully Convolutional Networks for Semantic Segmentation*<sup>[1]</sup> 中，作者选择了三个降维幅度，分别是 8 倍降维、16 倍降维和 32 倍降维，最终将这三个尺度的结果融合得到结果。

模型网络的大体结构已经确定，下面就要解决训练的问题了。既然分割问题转换成了分类问题，那么曾经训练过的分类模型就可以复用，分割模型可以在分类模型的基础上进行进一步的 finetune。在论文中作者选择了训练完成的 VGG 模型，将其改造成了适用于分割问题的模型，最终的模型结构如图 9-4 所示。

文章中还提到了一个有意思的问题，那就是采样学习的问题。前面介绍的分类问题都是一张图搭配一个类别信息，而现在的分割问题变成了一个像素对应一个类别信息。由于图像有局部相关性，相邻的像素和它对应的环境可能差别不大，而且它们同属于同一个类别，那么这么多十分近似的训练样本一起学习，会不会出现过拟合的问题？论文中给出的实验结果显示，过拟合的问题并不存在，但是当初作者表示担心，曾经想过用采样学习的方式解决这个问题。并不是最终输出的每一个像素的类别信息都会计算到梯度中进行反向传播，每幅图只取其中一部分的像素点。这个想法是有点道理的，如果每一个像素点都计算在内，而很多像素点又十分相似，那么它们相当于对某一个增加了很高的权重的样本。但好在如果对所有像素点都增加权重的话，这个影响还是会抵消的。

虽然模型采用分类的目标函数进行训练，但到了测试环节，模型还是要使用更适合分割问题的评估函数。对于经典的分类问题，常用的训练目标函数是 Cross Entropy Loss，模型评估则是判断最终的类别判断的正确率。在分割问题中，模型评估采用一个指标——IoU。它的全称是 Intersection of Union。这个算法利用了模型预测的边界和真实结果的边界，通过计算两个区域相交和它们相并面积的比例，就可以表示最终的分割效果。IoU 的结果范围在 (0, 1)，但模型追求的目标并不是完美的得分——1，其实只要拿到比较高的分数就说明物体的主体部分识别正确，那么分割也就基本正确了。IoU 的计算代码如下所示：

```
def calc_IoU(rectA, rectB):
```

```
    '''
```

```
    rect是一个长度为4的list，四个元素分别代表分割区域外接矩形的x坐标值，y
    坐标值，
    宽度和高度。
```

```
    '''
```

```
    xend_a = rectA[0] + rectA[2]
```

```
    yend_a = rectA[1] + rectA[3]
```

```
    xend_b = rectB[0] + rectB[2]
```

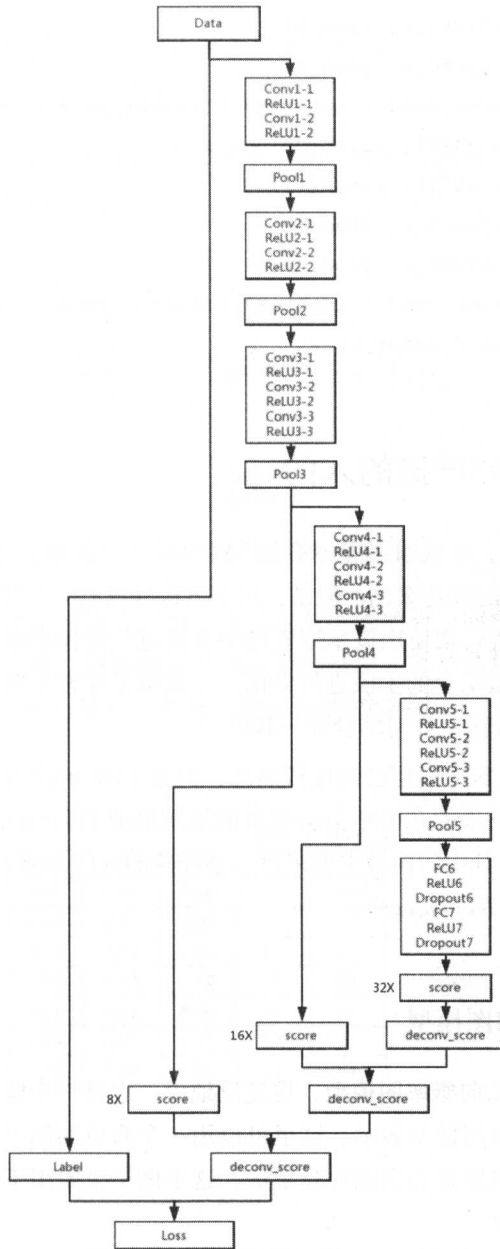


图 9-4 全卷积网络分割模型结构

```

yend_b = rectA[1] + rectA[3]
outer_x = min(rectA[0], rectB[0])
outer_y = min(rectA[1], rectB[1])

```

```

outer_xend = max(xend_a, xend_b)
outer_yend = max(yend_a, yend_b)
outer_area = (outer_xend - outer_x) * (outer_yend - outer_y)
inner_x = max(rectA[0], rectB[0])
inner_y = max(rectA[1], rectB[1])
inner_xend = min(xend_a, xend_b)
inner_yend = min(yend_a, yend_b)
inner_area = (inner_xend - inner_x) * (inner_yend - inner_y)
return inner_area / outer_area

```

## 9.2 CRF 通俗非严谨的入门

9.1 节基本上完成了对 FCN 的基本介绍。FCN 是一个将 High-level 问题的模型框架应用到 Low-level 问题的成功案例。但是，这个方法并没有完全解决问题。在深度学习火热前，图像分割问题经常使用概率图模型的方式进行建模求解，于是很多人开始尝试了 CNN 和 CRF 模型结合的手段进行尝试，并获得了非常不错的成绩。本章后面的篇幅里就来看看两种方法是如何结合在一起的。

相信各位读者对 CNN 模型已经比较熟悉，但是 CRF 的内容在本书前面的章节并未涉及，因此本书接下来的几个小节会尽可能地用最通俗直白的语言介绍 CRF 模型，为后面的内容做铺垫。其中的内容主要来自一本经典的入门书籍 *Probabilistic Graphical Models: Principles and Techniques*<sup>[2]</sup>。

### 9.2.1 有向图与无向图模型

CRF 模型是一个无向概率图模型，更宽泛地说，它是一个概率图模型。现实世界的一些问题可以用概率图模型表示。这里可以用一个简单的例子说明：建立一个简单的图模型来分析一部电影是否会获得高票房。这个例子主要用于介绍概率图模型，其中的观点内容纯属编造。

经过“认真”分析，我们发现一部电影的票房和以下因素有很大的关系。

- 剧本是否精彩，内容是否充实。
- 演员阵容是否强大，是否有可以吸引票房的明星。
- 演员表演是否精彩到位。
- 前期宣传是否到位。

- 上映时间是否合适，同期是否有其他实力强劲的电影。
- 投资。

上面的这些因素可以转化成一个个随机变量，将它们按照彼此之间的依赖关系进行连线，就得到了图 9-5。

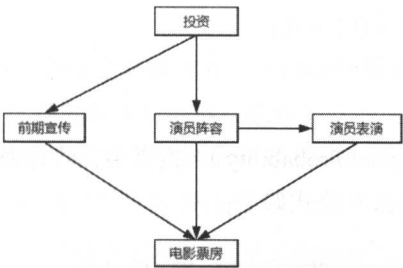


图 9-5 电影票房的概率图模型

从模型图中可以很清晰地看出每一个项目与电影票房之间的关系。拥有了这些关系，就可以根据其中一些变量推断出其他变量的情况。这里将每一个变量都离散化为 2 个等级——好和差，然后就可以根据分析得到的经验构建心目中的条件概率分布表，如图 9-6 所示。

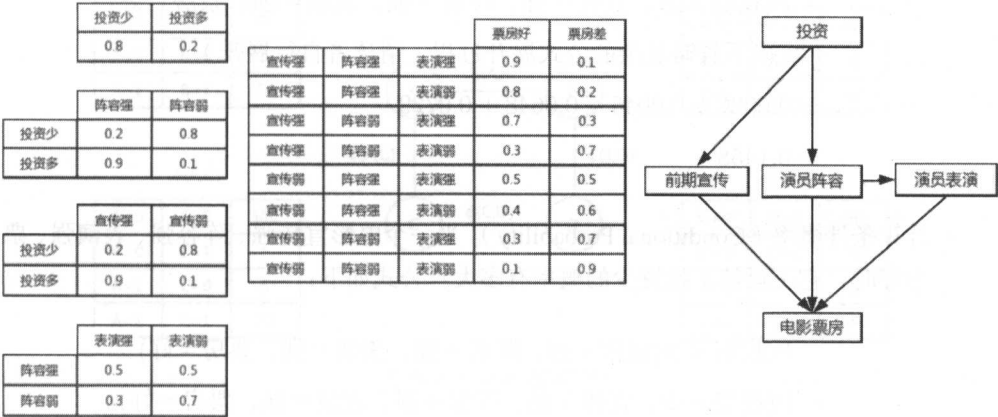


图 9-6 电影票房的概率图模型和模型具体参数

有了这个图和对应的表格，整个概率图模型就变得十分明确，这个图模型可以帮助完成很多事情，读者可以使用它在概率图模型的小世界中完成各种各样的推断，例如：

- 计算联合概率（Joint Probability）。一个投资少、宣传弱、阵容弱、表演弱且票房好的电影出现的概率有多大？公式如下：

$$\begin{aligned} &P(\text{投资} = \text{少}, \text{宣传} = \text{弱}, \text{阵容} = \text{弱}, \text{表演} = \text{弱}, \text{票房} = \text{好}) \\ &= P(\text{投资} = \text{少}) \times P(\text{宣传} = \text{弱} | \text{投资} = \text{少}) \times P(\text{阵容} = \text{弱} | \text{投资} = \text{少}) \\ &\times P(\text{表演} = \text{弱} | \text{阵容} = \text{弱}) \times P(\text{票房} = \text{好} | \text{宣传} = \text{弱}, \text{阵容} = \text{弱}, \text{表演} = \text{弱}) \\ &= 0.8 \times 0.8 \times 0.8 \times 0.7 \times 0.1 \\ &= 0.03584 \end{aligned}$$

- 计算边际概率 (Marginal Probability)。投资多，阵容强，票房好，其他无所谓的电影出现的概率有多大？公式如下：

$$\begin{aligned} &P(\text{投资} = \text{多}, \text{阵容} = \text{强}, \text{票房} = \text{好}) \\ &= \sum_{\text{宣传}} \sum_{\text{表演}} P(\text{投资} = \text{多}, \text{宣传}, \text{阵容} = \text{强}, \text{表演}, \text{票房} = \text{好}) \\ &= P(\text{投资} = \text{多}, \text{宣传} = \text{弱}, \text{阵容} = \text{强}, \text{表演} = \text{弱}, \text{票房} = \text{好}) \\ &+ P(\text{投资} = \text{多}, \text{宣传} = \text{弱}, \text{阵容} = \text{强}, \text{表演} = \text{强}, \text{票房} = \text{好}) \\ &+ P(\text{投资} = \text{多}, \text{宣传} = \text{强}, \text{阵容} = \text{强}, \text{表演} = \text{弱}, \text{票房} = \text{好}) \\ &+ P(\text{投资} = \text{多}, \text{宣传} = \text{强}, \text{阵容} = \text{强}, \text{表演} = \text{强}, \text{票房} = \text{好}) \\ &\quad (\text{以下省略复杂的公式展开过程，请读者自行展开}) \\ &= 0.0036 + 0.0045 + 0.0648 + 0.0729 \\ &= 0.1458 \end{aligned}$$

- 计算条件概率 (Conditional Probability)。当一个电影宣传强、阵容强、表演强、票房好时，它（竟然）投资少的概率有多大？公式如下：

$$\begin{aligned} &P(\text{投资} = \text{少} | \text{宣传} = \text{强}, \text{阵容} = \text{强}, \text{表演} = \text{强}, \text{票房} = \text{好}) \\ &= P(\text{投资} = \text{少}, \text{宣传} = \text{强}, \text{阵容} = \text{强}, \text{表演} = \text{强}, \text{票房} = \text{好}) \div \\ &\quad [P(\text{投资} = \text{少}, \text{宣传} = \text{强}, \text{阵容} = \text{强}, \text{表演} = \text{强}, \text{票房} = \text{好}) \\ &\quad + P(\text{投资} = \text{多}, \text{宣传} = \text{强}, \text{阵容} = \text{强}, \text{表演} = \text{强}, \text{票房} = \text{好})] \\ &= 0.1649 \end{aligned}$$

可以看出，这些推断都可以通过上面的图模型很好地推断出来，为理解这个模型提供更多的帮助。这就是一个简单的概率图模型的例子。当然，上面这个模型是一个有

向图模型，还不是 CRF 归属的无向图模型。

概率图模型主要有由有向图和无向图两部分组成。那么无向图和有向图有什么区别呢？就是随机变量的依赖关系。方向有什么好处和坏处呢？有了方向，整个概率图中概率或者信念（belief）的流动方向就可以确定，读者就能明白一个个随机变量之间的依赖关系，例如在上面的例子中，好几个因素和投资都有依赖关系，所以在求解时，投资这个因素需要首先明确。

在有向图模型中，每一个随机变量都拥有自己的条件概率分布（Conditional Probabilistic Distribution, CPD），这些随机变量的概率依赖于它的父辈随机变量的取值。这样的局部条件概率是很有用的，它使得计算联合概率和边际概率时变得比较容易。以上面的那些例子为例，在计算时我们只需要将这些 CPD 的取值连乘起来就可以了。

无向图因为没有方向，也就没有 CPD，但是无向图模型还是有自己的办法。无向图模型中同样的一个个类似 CPD 的东西被称作 **Factor**，像有向图中的节点拥有自己的 CPD 一样，Factor 也有自己的表示形式。它也可以像 CPD 一样用表格的形式表示。

例如空间内有四个粒子，每个粒子都有两种状态，它们之间还存在着一定的相互影响关系，这个关系由 Factor 来就如图 9-7 所示。

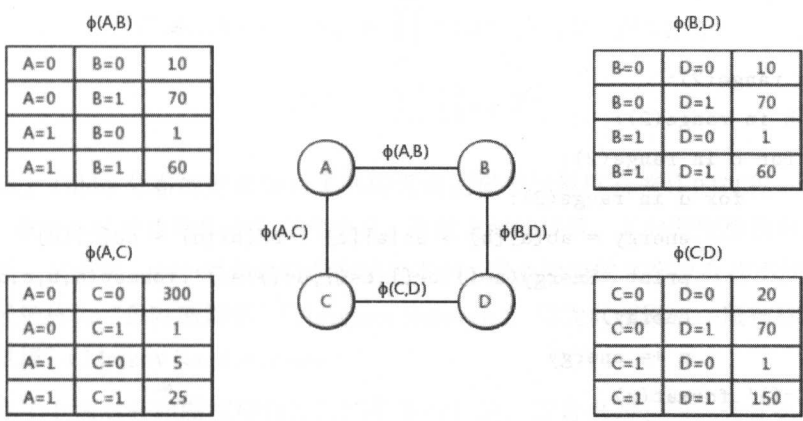


图 9-7 无向图例子

从上面的例子可以看出，Factor 和 CPD 相比有一个明显的不同。CPD 中所有的概率和为 1，而 Factor 里所有的 entry 没有和并不为 1。

和不为 1 对于读者并不是很好理解，概率的基本原则不就是需要所有事件发生的概率和为 1 吗？当然这没有错，因为无向图的 Factor 表示的并不是条件概率，而是一种更为对称的亲密关系（affinities）。求解无向图的联合概率需要换一种方法，那就是把所有的 Factor 像有向图模型的贝叶斯网络那样都连乘起来，再做一个归一化。



在上面的例子中，如果要求  $P(A = 1, B = 1, C = 1, D = 1)$  的概率，那么有：

$$\begin{aligned}
 & P(A = 1, B = 1, C = 1, D = 1) \\
 &= \frac{1}{Z} \phi(A = 1, B = 1) \times \phi(A = 1, C = 1) \times \phi(B = 1, D = 1) \times \phi(C = 1, D = 1) \\
 &= \frac{1}{Z} \times 60 \times 25 \times 60 \times 150 \\
 &= \frac{1}{Z} \times 60 \times 25 \times 60 \times 150 \\
 &= \frac{1}{Z} \times 13500000 \\
 &= 0.1128
 \end{aligned}$$

上面的计算也可以用代码的形式进行计算：

```

ab = [[10, 70], [1, 60]]
ac = [[300, 1], [5, 25]]
bd = [[10, 70], [1, 60]]
cd = [[20, 70], [1, 150]]

z = 0
for a in range(2):
    for b in range(2):
        for c in range(2):
            for d in range(2):
                energy = ab[a][b] * ac[a][c] * bd[b][d] * cd[c][d]
                print 'Energy(a={},b={},c={},d={})={}'.format(a,b,c,d,
                    energy)
                z += energy
print 'Z={}'.format(z)

```

这样就得到了所有的联合概率：

```

#以下为运行结果显示
Energy(a=0,b=0,c=0,d=0)=6000000
Energy(a=0,b=0,c=0,d=1)=14700000
Energy(a=0,b=0,c=1,d=0)=100
Energy(a=0,b=0,c=1,d=1)=105000
Energy(a=0,b=1,c=0,d=0)=420000
Energy(a=0,b=1,c=0,d=1)=88200000

```

```

Energy(a=0,b=1,c=1,d=0)=70
Energy(a=0,b=1,c=1,d=1)=630000
Energy(a=1,b=0,c=0,d=0)=1000
Energy(a=1,b=0,c=0,d=1)=24500
Energy(a=1,b=0,c=1,d=0)=250
Energy(a=1,b=0,c=1,d=1)=262500
Energy(a=1,b=1,c=0,d=0)=6000
Energy(a=1,b=1,c=0,d=1)=1260000
Energy(a=1,b=1,c=1,d=0)=1500
Energy(a=1,b=1,c=1,d=1)=13500000

```

从代码中可以看出，没有了有向图的依赖，无向图少了很多约束，计算公式反而更简洁。当完成归一化后，这些计算结果就可以像有向图那样表示随机变量的联合概率。这些联合概率实际上代表了无向图模型的概率分布，这种分布被称为 **Gibbs 分布**。Gibbs 分布就是利用 Factor 表示的无向图模型的概率分布，它的形式如下所示：

$$\begin{aligned}
 P(X_1, X_2, \dots, X_n) &= \frac{1}{Z(X)} \tilde{P}(X_1, X_2, \dots, X_n) \\
 \tilde{P}(X_1, X_2, \dots, X_n) &= \prod_{i=1}^m \phi_i(X) \\
 Z(X) &= \sum \prod \phi_i(X)
 \end{aligned}$$

实际上 Gibbs 分布的形式展示了利用无向图模型计算联合概率的过程。得到了联合概率，就可以计算边际概率和条件概率。通过上面的计算，无向图模型和有向图模型又走到同一起跑线。由于两者确实存在明显不同，因此两者的名字也有些不同，有向图网络一般被称为“贝叶斯网络”（Bayesian Network），而无向图网络一般被称为“马尔可夫随机场”（Markov Random Field）。

看完了上面对有向图模型和无向图模型的介绍，读者也许会问，为什么会有无向图和有向图这两类图模型，两者能不能合二为一？实际上这两种模型有各自的应用领域，有向图模型虽然清晰简单，但它并不能表示所有的真实场景，有向图模型通常需要一个有顺序的推断过程，这里暗示了一些依赖关系和独立条件，而无向图模型由于没有方向，也就没那么多限制，所以无向图模型可以用来对更多更复杂的问题进行建模。但是放弃了方向，也就意味着放弃了条件依赖和一些条件独立的特性，于是我们只能用 Factor 的形式和 Gibbs 分布进行表示，这个表示形式就有些复杂了。

除了上面介绍的区别，Factor 和 CPD 相比也有很大不同。因为没有和为 1 的限制，所以整体上看它对数值要求很宽松，但是它也有自己的坏处，那就是想从 Factor 的表

格形式中读出一些有价值的信息是比较困难的。这个困难有两个方面。

首先，因为不具有和为 1 的限制，无向图模型的概率比较抽象。关于这个问题读者去看几个真实的 Factor 表就能明白了。再看看贝叶斯网络的 CPD，就会感慨还是 CPD 描述得清楚。

其次，由于 Factor 的依赖关系不明朗，表格中记述的一些关系和全局状态下的关系有时是相反的。当读者具体观察某个 Factor 时，会觉得某组随机变量比另外一组亲密度高，产生的概率一定更高；但是如果站在全局观察，把联合概率计算出来再去计算它们的边际概率，就会发生 Factor 内表述的关系和全局信息相反。CPD 在这方面具有绝对优势，局部的条件概率放在全局还是合理的。

这里举一个例子，如果将上面的代码做一些改动，去求  $A, B$  的边际概率，就有：

```
for a in range(2):
    for b in range(2):
        energy = 0
        for c in range(2):
            for d in range(2):
                energy += ab[a][b] * ac[a][c] * bd[b][d] * cd[c][d]
            print 'Energy(a={},b={})={}'.format(a,b,energy)
#以下为运行结果显示
Energy(a=0,b=0)=15405100
Energy(a=0,b=1)=89250070
Energy(a=1,b=0)=288250
Energy(a=1,b=1)=14767500
```

和  $A, B$  所在的 Factor 相比， $\phi(A=1, B=1)$  在 Factor 中第二大，但是到了边际概率中它却成了第三大，说明从 Factor 中分析有时并不能看出某个事件的边际概率。

到此概率图模型和有向无向图模型相关的基本内容就介绍完了，相信读者对无向图模型有了一定的了解。下面就介绍更深入的概念。

### 9.2.2 Log-Linear Model

看完上面的无向图模型和 Gibbs 分布，读者实际上已经可以开始针对具体问题建模了，但是实际上上面介绍的表格形式的模型并不好用。为什么呢？采用表格的形式去表达模型，需要将随机变量的所有取值形式都写出来，如果变量的取值范围不大还可以接受，如果取值范围非常大，那么这种表格形式对我们建模来说就是个不小的负担，

所以表示 Factor 形式需要改变。为了解决这个问题，Factor 的形式需要被重新定义，首先需要把 Factor 函数转换成能量函数：

$$\phi(X) = \exp(-\xi(X))$$

$$\xi(X) = -\log(\phi(X))$$

我们把  $\phi(X)$  称作 Factor 函数，把  $\xi(X)$  称作能量函数 (Energy Function)。在物理学中，能量越大的物质存在的概率越小，能量越小的物质存在的概率越大。这个性质这组很符合函数的关系。这个函数带来了两个好处。

首先，Factor 函数中的每一项表示了随机变量间的亲密关系，一般来说这个值是非负的，这个限制会对建模造成困扰，因此利用指数函数变换，现在的 Energy 函数摆脱了非负数的限制，变得可正可负。

另外还有一个十分重要的特性：原来的乘法关系变成了现在的加法关系。我们现在有

$$\tilde{P}(X_1, X_2, \dots, X_n) = \exp\left(\sum_{i=k}^m \xi_i(X)\right)$$

变成加法关系后，建模求解都变得简单了不少，因为加法的关系更利于分析和计算。当然，模型形式变换到这一步还不够，想要得到进一步的化简，就要引入 Feature 这个概念。

读者已经了解了 Factor 的一般表现形式——表格的形式，但很多时候 Factor 的表格是比较稀疏的。虽然参与一个 Factor 的随机变量很多，但是真正有意义的亲密关系其实没几个。这样表格的形式就变得不再实用，Feature 表示的形式更适合这种场景，那么 Feature 形式是什么样的呢？

举个例子，有一个由两个灰白像素随机变量组成的 Factor，每个变量的取值范围为  $[0, 255]$  的整数。如果用 Factor 表示，这个 Table 将会有  $256 \times 256 = 65536$  个条目，但是如果这个 Factor 中表示的亲密关系和两个像素的值是否相等有关，像素值相等是关系为 1，不相等为 0，那么用 Feature 表示需要写成：

```
def feature(a,b):
    return 1 if a == b else 0
```

这种写法只要用两个条目就可以表述清楚。所以 Feature 就是通过尽可能地合并相同结果使 Factor 的表示变得简洁。以上就是 **Log-Linear** 模型的特点，可以看出它对无向图模型进行了极大的化简。未来的模型主要基于这一个框架进行构建。

### 9.2.3 条件随机场

条件随机场的全称是 Conditional Random Field (CRF)。它是马尔可夫随机场的一种特殊形式。前面说到了马尔可夫随机场和联合概率分布之间的计算关系，这里的条件随机场则主要对应了条件概率分布。条件随机场中参与计算的有两部分随机变量—— $X$  和  $Y$ 。一般来说， $X$  被称作观察变量，也就是已知的变量； $Y$  被称作目标变量或者隐含变量，是需要通过模型求解的变量。CRF 的出现与贝叶斯公式有关：

$$P(Y|X) = \frac{P(X, Y)}{P(X)}$$

其中  $P(X, Y)$  就是图模型的联合概率分布，而在有些问题中，对  $X$  单独建模十分困难，而对  $X, Y$  联合建模则相对容易些，这样的问题需要特殊的条件约束。条件随机场不允许任何一个 Factor 中只包含  $X$  的节点，Factor 中要么包含  $X$  和  $Y$ ，要么只包含  $Y$ 。于是上面的公式就可以做一定的修改，使得在建模时避免这些问题。

这里给出一种相对简单的条件随机场，如图 9-8 所示。

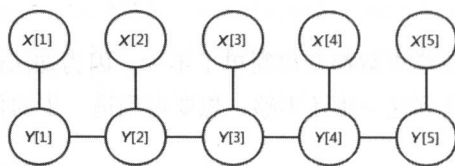


图 9-8 一种简单的条件随机场

它的形式如下所示：

$$P(Y|X) = \frac{1}{Z(X)} \tilde{P}(Y, X)$$

$$\tilde{P}(Y, X) = \exp\left(\sum_i w_i \cdot f_i(Y_t, X_t) + \sum_j w_j \cdot f_j(Y_t, Y_{t+1})\right)$$

$$Z(X) = \sum_Y \exp\left(\sum_i w_i \cdot f_i(Y_t, X_t) + \sum_j w_j \cdot f_j(Y_t, Y_{t+1})\right)$$

从图中可以观察到，它确实没有只包含  $X$  的 Factor。模型的条件概率通过计算联合概率和边际概率得到，而边际概率又是通过联合概率得到，这样难以建模的边际概率就通过联合概率得到解决。

采用无向图模型建模的 CRF 具有很强的表达能力和灵活性，但是计算起来却不那么容易。所有的概率推断必须从求解联合概率入手，还要计算非常复杂归一化项。所以

计算是无向图模型的一大难题，后面的内容会深入分析，解决难以计算这个问题。

## 9.3 Dense CRF

9.2 节介绍了概率图模型和 CRF 的基本概念，本节就来看看 CRF 在图像分割问题应用的具体形式——Dense CRF。这个模型来自论文 *Efficient Inference in Fully Connected CRFs with Gaussian Edge Potentials*<sup>[3]</sup>。CRF 中有两个关键变量，为了和领域内文献资料常用的变量名保持一致，本节要更换变量的名称——用  $X$  表示观察变量，也就是图像的像素信息，用  $Z$  表示隐含变量，也就是图像中每一个像素所属的类别，也就是要通过模型求解的信息。

### 9.3.1 Dense CRF 是如何被演化出来的

本节同样采用举例子的形式展现 Dense CRF 模型的演化过程，为了更好地展示即将提到的模型，这里给出一个简单的图形，这个丁老头的画像就是例子中分析的图像，如图 9-9 所示。

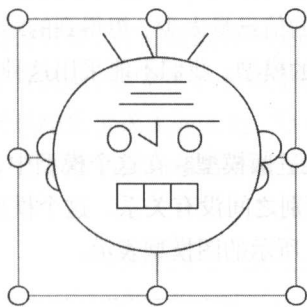


图 9-9 一个简单的示意图像

图像上的每一个小的等间距排列的小圆圈代表图像的像素  $x$ ，它们组成图像像素的集合  $X$ ；每一个像素所对应的隐式类别被称为  $z$ ，它们组成类别像素的集合  $Z$ ，如图 9-10 所示。

接下来就要根据图像和类别信息一步一步地完成从最简单的模型到复杂模型的构建。简单模型会以一种简单粗糙的方式看待图像像素与所表示的类别之间的关系，而复杂模型会以复杂细致的方式分析同一个问题。相信读者很快就会看出这其中的区别。

首先是最简单的建模方式 Plan A，假设每一个像素的类别只和自己所在的像素特征有关。这个模型可以用一个十分简单的图模型表示，如图 9-11 所示。

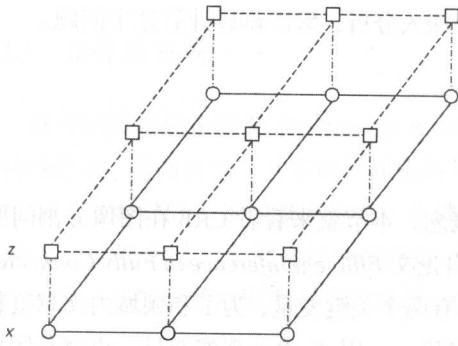


图 9-10 图像及对应标签组合

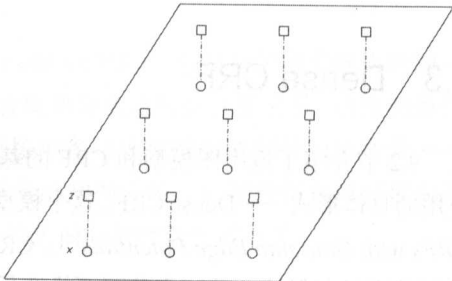


图 9-11 图像标签概率图模型方案 1

对这个模型来说，输入可以是一个像素的特征：像素的色彩值、位置等信息，输出是一个像素所代表的类别，也可以用一个向量表示，向量中每个位置表示这个像素属于当前位置类别的概率。这相当于模型对图像上的每一个像素单独进行一次分类操作：

$$P(Z|X) = \prod_i P(z_i|x_i)$$

这个模型的效果肯定不好，原因很简单——只通过一个像素就知道这个位置属于什么类别实在太难。像素本身的信息量太少，想得到的结果又太复杂。相信读者也可以一眼看出，它并不是一个靠谱的模型，我们不能采用这种方案模型，它只是我们前进路上的一个过渡模型。

下面再来看看 Plan B 的改进版模型。在这个模型中，每一个像素的类别和全体像素的图像信息有关，但像素类别之间没有关系。这个模型实际上就和 9.1 节中的 FCN 模型相同，模型可以用图 9-12 所示的图模型表示。

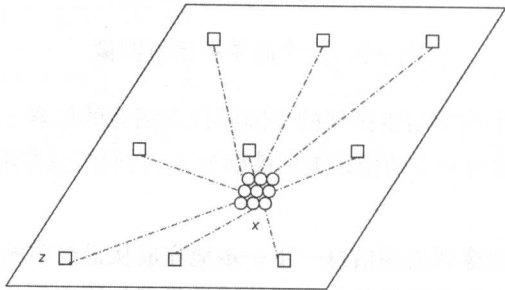


图 9-12 图像标签概率图模型方案 2

和 Plan A 相比，Plan B 已经有了很大进步。这一次要利用所有像素的信息判断一个位置的类别信息，而实际上像素信息并不需要这么多。当然这个模型也存在问题，FCN

模型只能解决图像像素和类别之间的相关关系，也就是对  $X$  和  $Z$  联合建模，而实际上每一个像素点类别之间也存在着关联关系，这就是图像领域常说的图像平滑性——每一个图像像素点的类别可能和临近点的类别很相近，这个特性是 FCN 模型所欠缺的，而这也是希望通过 CRF 模型弥补的。所以我们还不能停止脚步，还要继续前进探寻。

于是又一个新鲜出炉的新模型 Plan C 来了——这一次在 Plan B 的基础上，让每一个像素点的类别信息和它邻域的分类相关，模型得到进一步加强。于是它的图模型又变成了图 9-13 所示的样子。

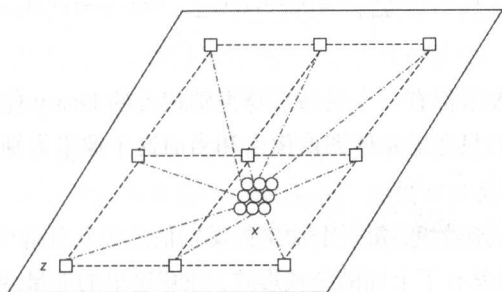


图 9-13 图像标签概率图模型方案 3

到这里像素类别之间的信息被融合进来，前面提到的平滑性也考虑在内。

上面的模型已经有了很大的突破，但有时这种程度的关联仍然是不够的，如果临近的几个像素恰好产生了一点异常波动，像素信息和预想的情况有很大不同，那么这个像素点的类别信息就有可能受到影响发生很大的变化。为了让模型变得更加鲁棒，模型还需要加入更多的关联关系。于是本章的主角模型 Plan D——Dense CRF 登场了。Dense CRF 的图模型如图 9-14 所示。

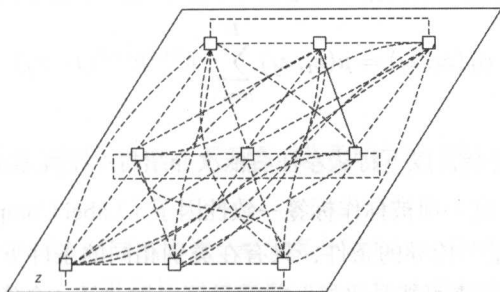


图 9-14 图像标签概率图模型方案 4(为了让图像不太混乱，其中  $x$  部分的内容已经省略)

除了保持 Plan C 中的连接关系，模型把任意一对像素的类别信息都连接上了，模型的复杂程度提到了最高的程度，现在看上去这个图就像一个线团，杂乱且不清晰。这



就是最终要选择的建模方式，完成了建模方案的确定，下面就用形式化的方法将它表示出来。

9.3.2 Dense CRF 的公式形式

Dense CRF 的能量函数表达形式如下：

$$E(z) = \sum_i \psi_u(X, z_i) + \sum_{i < j} \psi_p(X, z_i, z_j)$$

可以看出每一个像素都有一个只与自身类别相关的 Unary 能量函数，也就是说在这个特征函数里，我们只考虑全体图像像素和当前这个像素类别的关系，而不考虑别的像素的类别信息。

而后面的 Pairwise 函数里，每一个像素的类别信息都和其他像素的类别信息、所有像素的信息相关，于是就有了上面的公式形式。注意这里的能量函数是所有像素点的联合能量和，而不是在表示某一个像素点的能量。如果图像上有  $n$  个像素点，那么公式中就有  $n$  个 Unary 能量函数和  $\frac{n(n-1)}{2}$  个 Pairwise 能量函数。总的来说，特征函数数量是像素数量的平方级别，如果图片有 100 万像素，那么模型就会建立 4950 亿个 Pairwise 函数。正是因为这种复杂的形式，这个模型被称作 Dense CRF。图模型里满满的全是连线。

Unary 函数的内容一般比较灵活，Dense CRF 模型并没有对这一部分做详细的限制。因此这部分暂时略去不谈，把它放在后面介绍。下面将专注于介绍 Pairwise 函数。

根据论文中的内容，Pairwise 函数可以展开成如下形式：

$$\psi_p(z_i, z_j) = \mu(z_i, z_j) \sum_{m=1}^K w^{(m)} k^{(m)}(x_i, x_j)$$

这里的内容比较复杂，以下将从左往右依次介绍每一个元素。

首先是  $\mu(z_i, z_j)$ ，这一项被称作标签一致性因子（Label Compatibility Term），简单来说这里约束了“能量”传导的条件，只有在类别相同的条件下，能量才可以相互传导。具体来说，“一个像素可能是飞机”的能量可以和“另一个像素可能是飞机”的能量相互传导，从而增加或者减少后者“可能是飞机”的能量，进而影响“可能是飞机”的概率，而“一个像素可能是飞机”的能量是不能影响“另一个像素是人”的概率的。

当然文章中也提到，这样简单地根据标签硬性分离能量似乎有点不合理。拿 Pascal-VOC 训练集中的 20 个类别来说，有些类别之间的相似性很强，而另外一些类别则完全

不相似，所以论文作者认为不同类别间的能量转换可以以一定的损失率实现，不一定要完全拒绝。

接下来，公式中加和项里展示的就是经典的“权重乘以特征”的模型套路，其中特征可以展开成下面的公式：

$$k^{(m)}(f_i, f_j) = w^{(1)} \exp\left(-\frac{|p_i - p_j|^2}{2\theta_\alpha^2} - \frac{|I_i - I_j|^2}{2\theta_\beta^2}\right) + w^{(2)} \exp\left(-\frac{|p_i - p_j|^2}{2\theta_\gamma^2}\right)$$

这个公式以特征的形式表示了不同像素之间的“亲密度”。前面我们提到了特征形式比表格形式简洁，这个公式就可以很好地体现出来。公式中的第一项被称作 *Appearance Kernel*，第二项被称作 *Smooth Kernel*。这里面有很多变量，下面来依次分析。

*Appearance Kernel* 里面的  $p$  表示像素的位置——Position，图像是 2 维的，那么 Position 就有 2 维。 $I$  表示图像的像素值——Intensity，如果图像是彩色的，那么 Intensity 就有 3 维。分式下面的两个参数无论从位置还是长相都像高斯分布中的方差，这里的含义也差不多，用于表示模型对这些特征的敏感程度。

于是可以看出，如果两个像素距离近且颜色近，那么这个特征就会很强，反之则不强。同时分母也控制了对某一项目的敏感性，分母越小，模型对像素间的差异越敏感。其实这一项和图像处理中的双边滤波（*Bilateral Filter*）很像。这相当于每一个像素在一个 5 维的空间内寻找与自己相近的像素。

后面的 *Smooth Kernel* 和前面的 *Appearance Kernel* 类似。读者可以自行分析，这里不再赘述。

到这里 *Dense CRF* 的形式就介绍完了，毫无疑问，上面模型的复杂度实在太高，求解起来困难重重，模型需要进行适当化简才能轻松求解出来。这就要用到 9.4 节中将介绍的算法。

## 9.4 Mean Field 对 Dense CRF 模型的化简

9.3 节介绍了 *Dense CRF* 的基本形式，本节将介绍 *Dense CRF* 的求解思想——**Mean Field Variational Inference**（平均场变分推断）。这里涉及两个概念：*Mean Field* 和 *Variational Inference*。我们分开介绍这两个概念。

通过前面的学习，读者已经知道无向图模型中的一些类似概率的表示可以用 *Factor* 或者 *Feature* 表示。模型的目标是求出联合概率，并根据联合概率判断每一个像素类别的条件概率。在 *Dense CRF* 模型中，每一对像素之间都有一条边相连，每一条边都要

用一个 Factor 表示。模型太复杂，其中不确定的因素太多，通过 9.3 节介绍的公式直接求解非常困难。

由于 Dense CRF 的 Pairwise Factor 太多，如果一个像素点的 Energy 发生了变化，那么所有与它相连的像素点——当然就是剩下所有像素点的 Energy 都有可能随着这个像素点发生变化，于是可怕的蝴蝶效应就开始了。经过计算，在刚才提到的像素点中，一些像素点的 Energy 伴随着发生了变化，那么所有连着它们的像素点——几乎所有的像素点会跟着它们继续发生变动。这个过程会持续不断地进行，直到它们的 Energy 进入稳态，相互之间不再受影响这个过程才算结束。听上去这个能量传导会持续非常久。

实际上用能量讲述这个概念有些抽象，接下来将换一种说法：把每个连接一对像素点的 Factor 想象成一根弹簧，把像素点想象成可以有一定限度位移的小球。当一个小球稍微动了一下，所有连着它的弹簧会发生震动，那么弹簧另一边的小球也会跟着动，于是顺着弹簧，所有球都被传导起来，大家一起动起来。这个例子可能更有画面感，希望读者能感受到这个复杂模型的恐怖性。

面对如此复杂的模型，如果把这些 Energy 传导的过程都考虑在内，模型将变得非常难以求解。既然如此，这里就需要采用近似但快速的方法逼近最终答案。本节将对问题做适当的简化，在保证模型复杂度的同时使模型容易求解。

本节主要介绍 **Mean Field**。什么是 Mean Field 呢？以下将用一个形象且不严谨的方式理解这个概念。先给出结论，套用上面的例子，Mean Field approximation 将帮助模型把这个问题简化，使问题变成这个样子——小球的力量传导将简化成多个迭代，在每一轮迭代中，每一个小球将受到别的小球的弹簧力一次性计算完。其余的传导关系全部抛弃不管或放在下轮迭代中进行。

这个结论实在有些抽象，下面将用更详细具体的语言重新表达一遍。现在回到刚才描述的那个混乱的力传导场景，下面由上帝出马来调节这些弹簧的受力。所有受力由上帝支配。

第一步，小球如果受到了别的小球传导来的力，想把这个力传导给别人，那么它就可以提出一个申请，告诉上帝想传导出去的力。但是请注意，这只是一个申请，小球的力不会传递到别的小球上，上帝这里有一个小本本，她会把这些力先记下来，之后再按本子上记录的力量用她的魔法施加到这些弹簧上的。请大家放心，小球的力最终会足量地传递给别的小球，不会打折扣的，效果如图 9-15 所示。至于上帝为什么要用“她”，因为 God is a girl……

第二步，上帝施展分解大法，把整个模型拆解成许多个子模型，有多少个小球就有多少个子模型。在每一个子模型里，只有一个小球是主角，所有和这个小球相连的弹簧可以保留，所有和这个小球无关的弹簧全部被上帝拆掉。这段描述如图 9-16 所示。

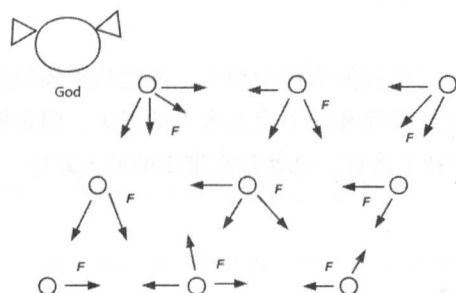


图 9-15 Mean Field 流程 1

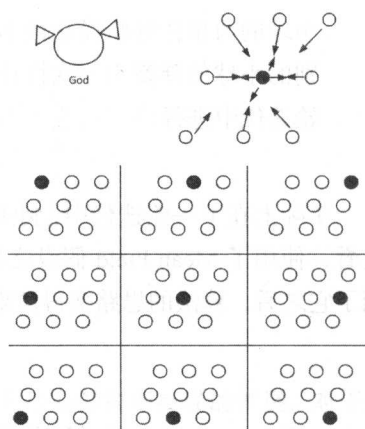


图 9-16 Mean Field 流程 2

第三步，上帝拿出刚才的小本本，把每个球的力施加到每一个子模型的弹簧上。于是每个子模型的主角——那个集万千弹簧于一身的小球受到了成吨的力，尽情地享受弹簧传来的力道吧！这段描述如图 9-17 所示。

第四步，这些主角在享受完弹簧传来的力道后，心想来而不往非礼也，也得传点力回去。这时上帝突然出现，放出大招，这些主角竟然无法把自己受到的力传回去。与此同时，在另一个平行世界——也就是别的子模型中，自己最初移动产生的力也已经传递给别的“主角”了。

第五步，上帝施展大法，将所有子模型合成原来的模型。这样一次力的传递过程就结束了，如图 9-18 所示。如果小球体内残存的洪荒之力还很多，就回到第一步，进行新一轮的能量传递。

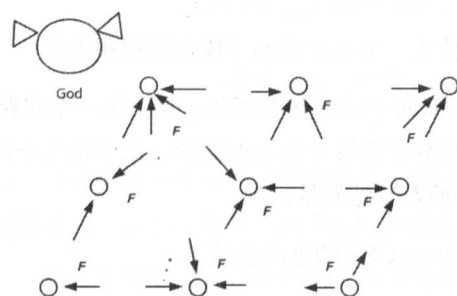


图 9-17 Mean Field 流程 3

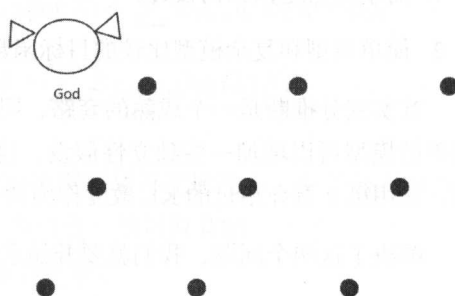


图 9-18 Mean Field 流程 4

这个力量传导的过程就从过去的复杂无序变成了有节奏、有规律的运动。读者可以再看看前面那句话：

小球的力量传导将简化成多个迭代，在每一轮迭代中，每一个小球将受到别的小球的弹簧力一次性计算完。其余的传导关系全部抛弃不管或放在下轮迭代中进行。

实际上在下一轮迭代中，小球依然要将自己受到的力传给别人，所以从整体效果上看，使用了 Mean Field 假设之后的模型和原始模型相比不会有太大的损失，但是使用了它之后，算法的思路变得清晰了许多，更利于运算，这就是它带来的最大好处。

## 9.5 Dense CRF 的推断计算公式

Mean Field 的假设帮助模型把能量传导问题变成了一个迭代更新的问题，但是它并没有改变无向图求解联合概率、计算最终结果的难题，求解联合概率依然是一个拥有很庞大计算量的工作，这对求解最终结果来说依然是个很大的麻烦。能不能进一步降低计算复杂度呢？答案当然是可以的。本节就来介绍求解要用到的另一个技巧：**Variational Inference**（变分推断）。

Variational Inference 方法的思想是使用一个简化后的模型代替原始复杂的模型，同时希望这个简化后的模型尽可能地靠近原来那个复杂的模型。如果两个模型完全一样，那就更好了。完成了模型的替代后，原本复杂的模型从此就可以扔掉不管，计算的中心将全部放在新模型上。

上面的原理介绍中有两个需要进一步解释的问题。

1. 简单的模型该如何设计？
2. 简单模型和复杂模型比较的目标函数是什么，怎么表示两个模型靠得很近？

其实变分推断是一个成熟的套路，回答上面两个问题并不复杂：对于第一个问题，简单的模型可以增加一些独立性假设，这样模型计算量又会大大减少；对于第二个问题，使用第 2 章介绍过的 KL 散度作为两个模型差异的衡量标准。

解决了这两个问题，我们就要开始求解 Dense CRF 的推断公式了。

本节将用大量的公式推导来解决这个复杂的计算过程。前面已经介绍过 Dense CRF 的能量函数，这里再和读者一起回顾下：

$$E(x) = \sum_i \psi_u(x_i) + \sum_{i < j} \psi_p(x_i, x_j)$$

对应的 Gibbs 分布如下所示：

$$P(X) = \frac{1}{Z} \tilde{P}(X) = \frac{1}{Z} \exp\left(-\sum_i \psi_u(x_i) - \sum_{i < j} \psi_p(x_i, x_j)\right)$$

为了变量表示方便， $z$  被换成  $x$ ，下面的推导将由此展开。

### 9.5.1 Variational Inference 推导

首先给出 KL 散度部分的推导，其实就是基于论文补充材料中推导的扩展。对于原问题的分布  $P(X)$ ，我们希望设计一个简单的分布模型  $Q(X)$ ，这个函数有一个很好的性质，那就是  $X$  在这个分布内是相互独立的，也就是说：

$$Q(x) = \prod_i Q_i(x_i)$$

根据这个性质可以给出两个分布的 KL 散度公式：

$$\begin{aligned} D(Q||P) &= \sum_x Q(x) \log\left(\frac{Q(x)}{P(x)}\right) \\ &= -\sum_x Q(x) \log P(x) + \sum_x Q(x) \log Q(x) \\ &= -E_{X \in Q}[\log P(X)] + E_{X \in Q}[\log Q(X)] \\ &= -E_{X \in Q}[\log \frac{1}{Z} \tilde{P}(X)] + E_{X \in Q}[\log \prod_i Q_i(X_i)] \\ &= -E_{X \in Q}[\log \tilde{P}(X)] + E_{X \in Q}[\log Z] + \sum_i E_{X_i \in Q}[\log Q_i(X_i)] \\ &= -E_{X \in Q}[\log \tilde{P}(X)] + \log Z + \sum_i E_{X_i \in Q_i}[\log Q_i(X_i)] \end{aligned}$$

为了使两个分布尽可能地靠近，KL 散度的值需要尽可能地变小。由于公式中的未知量是  $Q$ ，而公式的第二项  $\log Z$  中没有  $Q$ ，所以这一项可以省略。

除了上面的公式， $Q$  同时还需要满足下面的约束：

$$\sum_{x_i} Q_i(x_i) = 1$$

这种带约束的优化问题可以利用拉格朗日乘子法将两个公式融合在一起，这样就

得到：

$$L(Q_i) = -E_{X_i \in Q}[\log \tilde{P}(X)] + \sum_i E_{x_i \in Q_i}[\log Q_i(x_i)] + \lambda(\sum_{x_i} Q_i(x_i) - 1)$$

如果能对这个公式求导，并得到导数为 0 处的结果，就可以得到想要的分布  $Q$ 。这个公式的后面两项求导相对比较简单，前面的第一项比较复杂，首先对它单独做一下处理：

$$-E_{X_i \in Q}[\log \tilde{P}(X)] = -\int \prod_i Q_i(x_i) [\log \tilde{P}(X)] dX$$

接下来需要将变量做分离，这里将所有的变量  $X$  分成待求的  $x$  和其他变量  $\bar{X}$ ，于是就有：

$$\begin{aligned} &= -\int Q_i(x_i) \prod_i Q(\bar{x}_i) [\log \tilde{P}(X)] dx_i d\bar{X} \\ &= -\int Q_i(x_i) E_{\bar{X} \in Q}[\log \tilde{P}(X)] dx_i \end{aligned}$$

经过上面的公式整理，求出公式的偏导数，可得：

$$\frac{\partial L(Q_i)}{\partial Q_i(x_i)} = -E_{\bar{X} \in Q_i}[\log \tilde{P}(X|x_i)] + \log Q_i(x_i) + 1 + \lambda$$

令偏导为 0，就可以求出函数的表示形式：

$$Q_i(x_i) = \exp(-\lambda - 1) \exp(E_{\bar{X} \in Q_i}[\log \tilde{P}(X|x_i)])$$

由于每一个  $Q$  函数的  $\exp(-\lambda - 1)$  项都相同，这里将其当作一个常数项，当像素点所有类别的概率计算完成后再进行归一化将其抵消掉，于是  $Q$  函数就等于：

$$Q(x_i) = \frac{1}{Z_1} \exp(E_{\bar{X} \in Q_i}[\log \tilde{P}(X|x_i)])$$

再将关于  $\tilde{P}$  的定义带入，就得到：

$$Q(x_i) = \frac{1}{Z_1} \exp(E_{\bar{X} \in Q}[( - \sum_i \psi_u(x_i) - \sum_{j \neq i} \psi_p(x_i, x_j)) | x_i])$$

由于这里的  $x_i$  是已知的，所以可以得到论文中补充材料里的结论（但是变量名不

太一样)：

$$Q_i(x_i = l) = \frac{1}{Z_i} \exp[-\psi_u(l) - \sum_{j \neq i} E_{\bar{X} \in Q_j} \psi_p(l, X_j)]$$

到这里终于得到了这个简单的模型，现在计算某个像素点的类别概率时，其他点将会固定不变，复杂的联合概率终于被抛在了脑后。

### 9.5.2 进一步化简

虽然经过 Mean Field Variational Inference 的化简，计算公式已经简单了许多，但是公式中的计算量依然不少。同样的想法再次出现，计算能不能算得再简化一些？前面看到 Dense CRF 在计算 Pairwise Energy 和双边滤波算法时十分接近，而双边滤波是可以有更简单的计算方法的，那能不能顺着这个思路对模型进行进一步化简呢？顺着这个思路，继续对上面得到的公式做进一步展开和变换：

$$\begin{aligned} &= \frac{1}{Z_i} \exp[-\psi_u(l) - \sum_{m=1}^K w^{(m)} \sum_{j \neq i} E_{X \in Q_j} [\mu(l, X_j) k^{(m)}(f_i, f_j)]] \\ &= \frac{1}{Z_i} \exp[-\psi_u(l) - \sum_{m=1}^K w^{(m)} \sum_{j \neq i} \sum_{l' \in L} Q_j(l') \mu(l, l') k^{(m)}(f_i, f_j)] \\ &= \frac{1}{Z_i} \exp[-\psi_u(l) - \sum_{l' \in L} \mu(l, l') \sum_{m=1}^K w^{(m)} \sum_{j \neq i} Q_j(l') k^{(m)}(f_i, f_j)] \end{aligned}$$

这样，通过简单的变换，我们达到了我们的目标，其中最内层的加权求和可以用截断的高斯滤波完成。搬运论文中最后的一点公式，可以得：

$$Q_i^{(\tilde{m})}(l) = \sum_{j \neq i} Q_j(l') k^{(m)}(f_i, f_j) = \sum_j Q_j(l) k^{(m)}(f_i, f_j) - Q_i(l)$$

上面公式的最后一项可以转化成卷积操作，而卷积操作又有一些加速运算的方法。经过不懈地努力，问题终于被转化到读者熟悉的领域——卷积。下面给出 Dense CRF 完整的算法，也算是对这本节的总结。

---

#### Mean Field Dense CRF 算法

1. 利用 Unary Energy 函数初始化  $Q_i(x_i) = \frac{1}{Z_i} e^{-\phi_u(x_i)}$ 。
2. 如果没有收敛，则反复执行下面的内容。



- $\tilde{Q}_i^{(m)}(l) = \sum_{j \neq i} k^{(m)}(f_i, f_j) Q_j(l)$  for all m
- $\hat{Q}_i(x_i) = \sum_{l \in L} \mu^{(m)}(x_i, l) \sum_m w^{(m)} \tilde{Q}_i^{(m)}(l)$
- $Q_i(x_i) = e^{-\psi_u(x_i) - \tilde{Q}_i(x_i)}$
- $Q_i(x_i) = \frac{1}{Z} Q_i(x_i)$
- 判断是否收敛，如果收敛，则算法结束

模型从最初复杂的样子一步步化简到现在一个看上去可以求解的结果，似乎已经得到了完美的结果，但是到了这一步我们还是不禁要问一句，这个计算过程可不可以再做一些变换，让计算变得更简单，让 Dense CRF 和 FCN 模型结合起来呢？

## 9.6 完整的模型：CRF as RNN

本节将把前面介绍的两个模型串联起来，继承一个完整的解决方案。两个模型结合起来的方案如下所示。

1. 前面在介绍 Dense CRF 时曾提到 Unary Energy Function 的设计比较灵活，于是这里就让 FCN 的结果作为 Unary Function 的结果。
2. 让 FCN 的结果作为 Pairwise Function 中的 Q 函数的初始值。

这样 FCN 和 CRF 就连起来了。模型的表示问题解决了，下面将把目标转向实现的问题。Caffe 处理实现 FCN 模型，那么 Dense CRF 呢？它需要单独实现吗？还是把它也嵌入 Caffe 的框架里？*Conditional Random Fields as Recurrent Neural Networks*<sup>[4]</sup> 论文的作者选择了第二种方案。

在这篇讲述将 FCN 和 Dense CRF 两个模型结合的论文中，作者将 CRF 的求解过程转换成了 RNN 的形式。同时，由于 CRF 的求解算法是迭代进行的，因此算法可以展开成 RNN 的形式，模型被称为 CRF as RNN。接下来就简单地看看这个转换的细节。论文的源代码地址为 <https://github.com/torrvision/crfasrnn>。

这个模型的结构图并不复杂，由于 FCN 模型前面已经介绍过了，这里的模型只是它的基础上增加了 Dense CRF 的模型层，模型结构如图 9-19 所示。

下面来重点分析最后一层的 MULTI\_STAGE\_MEANFIELD，虽然在文章中作者提到了 CRF as RNN 的概念，但是实际上它的实现并没有直接使用 RNN 的框架，而是写成了专用的模型层。由于模型层将所有的 CRF 计算内容集成到了一起，所以这一层会比较复杂，计算量也比较大，其中的反向传播也比较复杂。

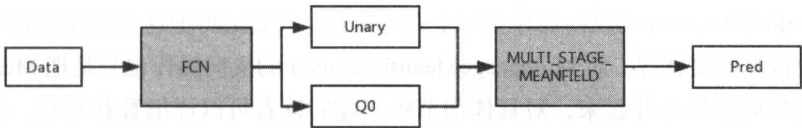


图 9-19 CRFasRNN 模型的结构

计算主要涉及两个类别——MultiStageMeanfieldLayer 和 MeanfieldIteration，其中 MultiStageMeanfieldLayer 类主要负责算法内容的组织，而 MeanfieldIteration 类主要负责迭代计算过程。这部分算法的流程图如图 9-20 和图 9-21 所示。

MultiStageMeanfieldLayerForward

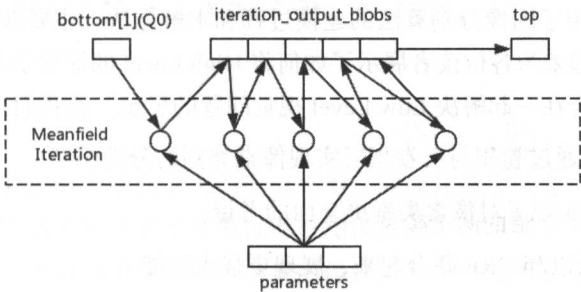


图 9-20 MultiStageMeanfieldLayer 的迭代过程

Meanfield Iteration Inner Forward

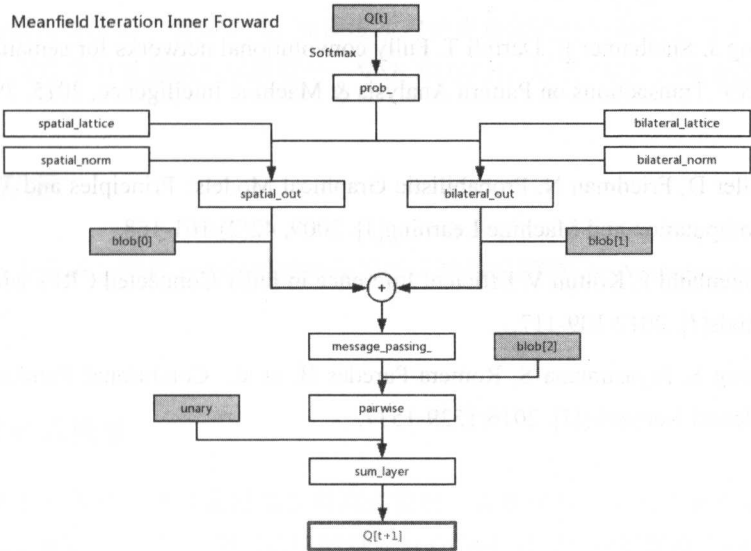


图 9-21 Meanfield Iteration 计算过程，图中有底纹的框表示迭代的输入和参数，加粗的框表示输出

MeanfieldIteration 的反向操作实际上并不复杂，只要记得把这些过程拆解成一个个小部分慢慢算梯度就好，而 MultiStageMeanfieldLayer 的反向操作也只是把 MeanfieldIteration 的反向结果合并起来，对具体细节感兴趣的读者可以详细看看源码，这里就不再赘述了。

到这里，我们已经实现了将 CNN 和 CRF 无缝连接起来了。比起之前的单独由 CNN 组成的 FCN，模型的性能有了很大的提升。

## 9.7 总结

本章主要介绍了图像分割算法的建模过程和求解方法。主要涉及多篇论文的研究成果，它们联合起来为各位读者展示了如何将 High Level 的深度学习模型和 Low Level 的概率图模型结合在一起解决 Low Level 视觉问题的思想。让我们回顾一下。

- 全卷积网络通过卷积与反卷积层实现像素类别的分类模型。
- Dense CRF 实现了对像素类别更全面的考量。
- FCN 模型可以和 CRF 联合起来，展现更强大的能力。

## 9.8 参考文献

[1] Long J, Shelhamer E, Darrell T. Fully convolutional networks for semantic segmentation[J]. IEEE Transactions on Pattern Analysis & Machine Intelligence, 2015, 79(10):1337-1342.

[2] Koller D, Friedman N. Probabilistic Graphical Models: Principles and Techniques - Adaptive Computation and Machine Learning[J]. 2009, 42(2):161-168.

[3] Krähenbühl P, Koltun V. Efficient Inference in Fully Connected CRFs with Gaussian Edge Potentials[J]. 2012:109-117.

[4] Zheng S, Jayasumana S, Romera-Paredes B, et al. Conditional Random Fields as Recurrent Neural Networks[J]. 2016:1529-1537.

## 应用：图像生成

本章将为读者介绍基于深度学习的生成模型。前面几章主要介绍了机器学习中的判别式模型，这种模型的形式主要是根据原始图像推测图像具备的一些性质，例如根据数字图像推测数字的名称，根据自然场景图像推测物体的边界；而生成模型恰恰相反，通常给出的输入是图像具备的性质，而输出是性质对应的图像。这种生成模型相当于构建了图像的分布，因此利用这类模型，我们可以完成图像自动生成（采样）、图像信息补全等工作。在深度学习之前已经有很多生成模型，但苦于生成模型难以描述难以建模，科研人员遇到了很多挑战，而深度学习的出现帮助他们解决了不少问题。本章就介绍基于深度学习思想的生成模型——VAE 和 GAN，以及 GAN 的变种模型。

### 10.1 VAE

本节将为读者介绍基于变分思想的深度学习的生成模型——Variational autoencoder，简称 VAE<sup>[1]</sup>。

#### 10.1.1 生成式模型

前面的章节里读者已经看过很多判别式模型。这些模型大多有下面的规律：已知观察变量  $X$ ，和隐含变量  $z$ ，判别式模型对  $p(z|X)$  进行建模，它根据输入的观察变量  $x$  得到隐含变量  $z$  出现的可能性。生成式模型则是将两者的顺序反过来，它要对  $p(X|z)$  进行建模，输入是隐含变量，输出是观察变量的概率。

可以想象，不同的模型结构自然有不同的用途。判别模型在判别工作上更适合，生成模型在分布估计等问题上更有优势。如果想用生成式模型去解决判别问题，就需要利用贝叶斯公式把这个问题转换成适合自己处理的样子：

$$p(z|X) = \frac{p(X|z)p(z)}{p(X)}$$

对于一些简单的问题，上面的公式还是比较容易解出的，但对于一些复杂的问题，找出从隐含变量到观察变量之间的关系是一件很困难的事情，生成式模型的建模过程会非常困难，所以对于判别类问题，判别式模型一般更适合。

但对于“随机生成满足某些隐含变量特点的数据”这样的问题来说，判别式模型就会显得力不从心。如果用判别式模型生成数据，就要通过类似于下面这种方式的方法进行。

第一步，利用简单随机一个  $X$ 。

第二步，用判别式模型计算  $p(z|X)$  概率，如果概率满足，则找到了这个观察数据，如果不满足，返回第一步。

这样用判别式模型生成数据的效率可能会十分低下。而生成式模型解决这个问题就十分简单，首先确定好  $z$  的取值，然后根据  $p(X|z)$  的分布进行随机采样就行了。

了解了两种模型的不同，下面就来看看生成式模型的建模方法。

### 10.1.2 Variational Lower bound

虽然生成模型和判别模型的形式不同，但两者建模的方法总体来说相近，生成模型一般也通过最大化后验概率的形式进行建模优化。也就是利用贝叶斯公式：

$$p(z|X) = \frac{p(X|z)p(z)}{\int_z p(X|z)p(z)dz}$$

这个公式在复杂的模型和大规模数据面前极难求解。为了解决这个问题，这里将继续采用变分的方法用一个变分函数  $q(z)$  代替  $p(z|X)$ 。第 9 章在介绍 Dense CRF 时已经详细介绍了变分推导的过程，而这一次的推导并不需要做完整的变分推导，只需要利用变分方法的下界将问题进行转换即可。

既然希望用  $q(z)$  这个新函数代替后验概率  $p(z|X)$ ，那么两个概率分布需要尽可能

地相近，这里依然选择 KL 散度衡量两者的相近程度。根据 KL 公式就有：

$$\begin{aligned}\text{KL}(q(z)||p(z|X)) &= \int q(z) \log \frac{q(z)}{p(z|X)} dz \\ &= \int q(z) [\log q(z) - \log p(z|X)] dz\end{aligned}$$

根据贝叶斯公式进行变换，就得到了：

$$\begin{aligned}&= \int q(z) [\log q(z) - \log \frac{p(X|z)p(X)}{p(z)}] dz \\ &= \int q(z) [\log q(z) - \log p(X|z) - \log p(z) + \log p(X)] dz\end{aligned}$$

由于积分的目标是  $z$ ，这里再将和  $z$  无关的项目从积分符号中拿出来，就得到了：

$$= \int q(z) [\log q(z) - \log p(X|z) - \log p(z)] dz + \log p(X)$$

将等式左右项目交换，就得到了下面的公式：

$$\log p(X) - \text{KL}(q(z)||p(z|X)) = \int q(z) \log p(X|z) dz - \text{KL}(q(z)||p(z))$$

虽然这个公式还是很复杂，因为 KL 散度的性质，这个公式中还是令人看到了一丝曙光。

首先看等号左边，虽然  $p(X)$  的概率分布不容易求出，但在训练过程中当  $X$  已经给定， $p(X)$  已经是个固定值不需要考虑。如果训练的目标是希望  $\text{KL}(q(z)||p(z|X))$  尽可能小，就相当于让等号右边的那部分尽可能变大。等号右边的第一项实际上是基于  $q(z)$  概率的对数似然期望，第二项又是一个负的 KL 散度，所以我们可以认为，为了找到一个好的  $q(z)$ ，使得它和  $p(z|X)$  尽可能相近，实现最终的优化目标，优化的目标将变为：

- 右边第一项的 log 似然的期望最大化：

$$\max \int q(z) \log p(X|z) dz$$

- 右边第二项的 KL 散度最小化：

$$\min \text{KL}(q(z)||p(z))$$

右边两个项目的优化难度相对变小了一些，下面就来看看如何基于它们做进一步的计算。

### 10.1.3 Reparameterization Trick

为了方便地求解上面的公式，这里需要做一点小小的 trick 工作。上面提到了  $q(z)$  这个变分函数，为了近似后验概率，它实际上代表了给定某个  $X$  的情况下  $z$  的分布情况，如果将它的概率形式写完整，那么它应该是  $q(z|X)$ 。这个结构实际上对后面的运算产生了一些障碍，那么能不能想办法把  $X$  抽离出来呢？

例如，有一个随机变量  $a$  服从均值为 1，方差为 1 的高斯分布，那么根据高斯分布的性质，随机变量  $b = a - 1$  将服从均值为 0，方差为 1 的高斯分布，换句话说，我们可以用一个均值为 0，方差为 1 的随机变量加上一个常量 1 来表示现在的随机变量  $a$ 。这样一个随机变量就被分成了两部分——一部分是确定的，一部分是随机的。

实际上， $q(z|X)$  也可以采用上面的方法完成。这个条件概率可以拆分成两部分，一部分是一个观察变量  $g_\phi(X)$ ，它代表了条件概率的确定部分，它的值和一个随机变量的期望值类似；另一部分是随机变量  $\epsilon$ ，它负责随机的部分，基于这样的表示方法，条件概率中的随机性将主要来自这里。

这样做有什么好处呢？经过变换，如果  $z$  条件概率值完全取决于  $\epsilon$  的概率。也就是说如果  $z^{(i)} = g_\phi(X + \epsilon^{(i)})$ ，那么  $q(z^{(i)}) = p(\epsilon^{(i)})$ ，那么上面关于变分推导的公式就变成了下面的公式：

$$\log p(X) - \text{KL}(q(z)||p(z|X)) = \int p(\epsilon) \log p(X|g_\phi(X, \epsilon)) d\epsilon - \text{KL}(q(z|X, \epsilon)||p(z))$$

这就是替换的一小步，求解的一大步！这个公式已经很接近问题最终的答案了，既然  $\epsilon$  完全决定了  $z$  的分布，那么假设一个  $\epsilon$  服从某个分布，这个变分函数的建模就完成了。如果  $\epsilon$  服从某个分布，那么  $z$  的条件概率是不是也服从这个分布呢？不一定。 $z$  的条件分布会根据训练数据进行学习，由于经过了函数  $g_\phi()$  的计算， $z$  的分布有可能产生了很大的变化。而这个函数，就可以用深度学习模型表示。前面的章节读者已经了解到深层模型的强大威力，那么从一个简单常见的随机变量映射到复杂分布的变量，对深层模型来说是一件很平常的事情，它可以做得很好。

于是这个假设  $\epsilon$  服从多维且各维度独立高斯分布。同时， $z$  的先验和后验也被假设成一个多维且各维度独立的高斯分布。下面就来看看两个优化目标的最终形式。

### 10.1.4 Encoder 和 Decoder 的计算公式

回顾一下 10.1.2 的两个优化目标，下面就来想办法求解这两个目标。首先来看看第二个优化目标，也就是让公式右边第二项  $KL(q(z)||p(z))$  最小化。刚才  $z$  的先验被假设成一个多维且各维度独立的高斯分布，这里可以给出一个更强的假设，那就是这个高斯分布各维度的均值为 0，协方差为单位矩阵，那么前面提到的 KL 散度公式就从：

$$KL(p1||p2) = \frac{1}{2} [\log \frac{\det(\Sigma_2)}{\det(\Sigma_1)} - d + \text{tr}(\Sigma_2^{-1}\Sigma_1) + (\mu_2 - \mu_1)^T \Sigma_2^{-1} (\mu_2 - \mu_1)]$$

瞬间简化成为：

$$KL(p1||N(0, I)) = \frac{1}{2} [-\log[\det(\Sigma_1)] - d + \text{tr}(\Sigma_1) + \mu_1^T \mu_1]$$

前面提到了一个用深层网络实现的模型  $g_\phi(X, \epsilon)$ ，它的输入是一批图像，输出是  $z$ ，因此这里需要它通过  $X$  生成  $z$ ，并将这一个批次的数据汇总计算得到它们的均值和方差。这样利用上面的公式，KL 散度最小化的模型就建立好了。

实际计算过程中不需要将协方差表示成矩阵的形状，只需要一个向量  $\sigma_1$  来表示协方差矩阵的主对角线即可，公式将被进一步简化：

$$KL(p1(\mu_1, \sigma_1)||N(0, I)) = \frac{1}{2} [-\sum_i \log[(\sigma_{1i})] - d + \sum_i (\sigma_{1i}) + \mu_1^T \mu_1]$$

由于函数  $g_\phi()$  实现了从观测数据到隐含数据的转变，因此这个模型被称为 Encoder 模型。

接下来是第一个优化目标，也就是让公式左边第一项的似然期望最大化。这一部分的内容相对简单，由于前面的 Encoder 模型已经计算出了一批观察变量  $X$  对应的隐含变量  $z$ ，那么这里就可以再建立一个深层模型，根据似然进行建模，输入为隐含变量  $z$ ，输出为观察变量  $X$ 。如果输出的图像和前面生成的图像相近，那么就可以认为似然得到了最大化。这个模型被称为 Decoder，也就是本章的主题——生成模型。

到这里 VAE 的核心计算推导就结束了。由于模型推导的过程有些复杂，下面就来看看 VAE 实现的代码，同时来看看 VAE 模型生成的图像是什么样子。



10.1.5 实现

本节要介绍 VAE 模型的一个比较不错的实现——GitHub - cdoersch/vae\_tutorial: Caffe code to accompany my Tutorial on Variational Autoencoders([https://link.zhihu.com/?target=https%3A//github.com/cdoersch/vae\\_tutorial](https://link.zhihu.com/?target=https%3A//github.com/cdoersch/vae_tutorial)), 这个工程还配有一个介绍 VAE 的文章 [2], 感兴趣的读者可以阅读, 读后会有更多启发。这个实现使用的目标数据集依然是 MNIST, 模型的架构如图 10-1 所示。为了更好地了解模型的架构, 这里将模型中的一些细节隐去, 只留下核心的数据流动和 Loss 计算部分。

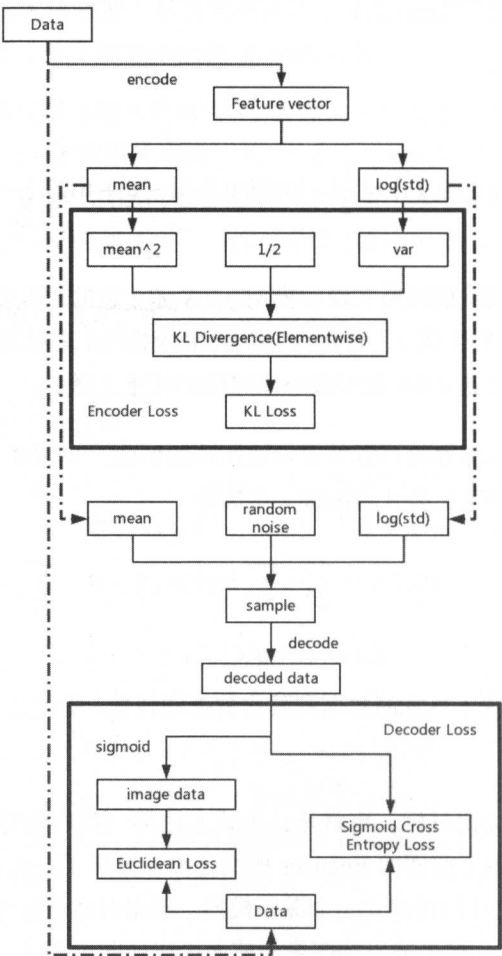


图 10-1 VAE 模型结构图

图中粗框表示求解 Loss 的部分。虚线展现了两个模块之间数据共享的情况。可以看出图的上半部分是优化 Encoder 的部分, 下面是优化 Decoder 的部分, 除了 Encoder

和 Decoder，图中还有三个主要部分。

- Encoder 的 Loss 计算：KL 散度。
- $z$  的重采样生成。
- Decoder 的 Loss 计算：最大似然。

这其中最复杂的就是第一项，Encoder 的 Loss 计算。由于 Caffe 在实际计算过程中只能采用向量的计算方式，没有广播计算的机制，所以前面的公式需要进行一定的变换：

$$\begin{aligned} \text{KL}(p_1(\mu_1, \sigma_1) || N(0, I)) &= \frac{1}{2} \left[ - \sum_i \log[(\sigma_{1i})] - d + \sum_i (\sigma_{1i}) + \mu_1^T \mu_1 \right] \\ &= \sum_{i=0}^d -\frac{1}{2} \log[\text{std}_{1i}^2] + \sum_{i=0}^d \left(-\frac{1}{2}\right) + \sum_{i=0}^d \frac{1}{2} (\text{std}_{1i}^2) + \sum_{i=0}^d \frac{1}{2} [\mu_{1i}^2] \\ &= \sum_{i=0}^d \left[ -\frac{1}{2} \log[\text{std}_{1i}^2] + \frac{1}{2} (\text{std}_{1i}^2) + \frac{1}{2} [\mu_{1i}^2] + \left(-\frac{1}{2}\right) \right] \end{aligned}$$

在完成了前面的向量级别计算后，最后一步就是完成汇总加和的过程。这样 Loss 计算就顺利完成了。

经过上面对 VAE 理论和实验的介绍，相信读者对 VAE 模型有了更清晰的认识。经过训练后 VAE 的解码器在 MNIST 数据库上生成的字符如图 10-2 所示。

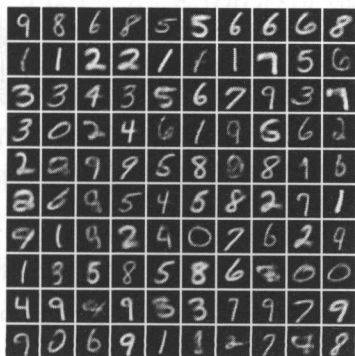


图 10-2 VAE 生成的数字图

### 10.1.6 MNIST 生成模型可视化

除了直接观察最终生成的数字结果，实际上还有另一种观察数据的方式，那就是站在隐变量空间的角度观察分布的生成情况。实现这个效果需要完成以下两个工作。

- 1. 隐变量的维度为 2，相当于把生成的数字图片投影到 2 维平面上，这样更方便可视化观察分析。
- 2. 由于隐变量的维度为 2，就可以从二维平面上等间距地采样一批隐变量，这样这批隐变量可以代表整个二维平面上隐变量的分布，然后这批隐变量经过解码器处理后展示，这样就可以看到图像的分布情况了。

上面描述的算法的流程如图 10-3 所示。

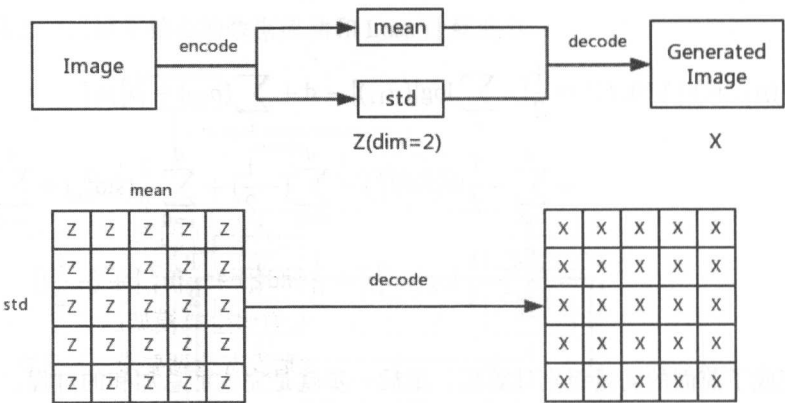


图 10-3 模型可视化流程图。上图主要标识了模型的修改部分，下图介绍隐变量采样和生成的形式

图 10-4 所示的模型很好地完成了隐变量的建模，绝大多数数字出现在了平面分布中，数字与数字一些过渡区域，这些过渡区域的图像拥有多个数字的特征，而这些都是数字的外形确实存在着相似之处。可以明显地感受到，图像随着隐变量变换产生了变换。

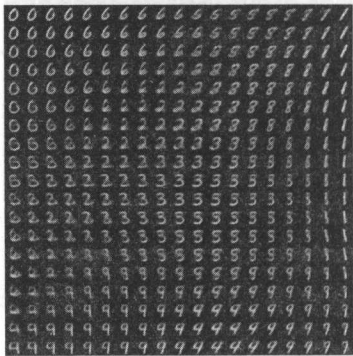


图 10-4 模型可视化结果

VAE 的内容就介绍到这里，下面来看看另一个生成模型。

## 10.2 GAN

前面我们介绍了 VAE，下面来看看 GAN (Generative Adversarial Network) [3]，这个网络组是站在对抗博弈的角度展现生成模型和判别模型各自的威力的，在效果上比 VAE 还要好些。

### 10.2.1 GAN 的概念

同 VAE 模型类似，GAN 模型也包含了一对子模型。GAN 的名字中包含一个对抗的概念，为了体现对抗这个概念，除了生成模型，其中还有另外一个模型帮助生成模型更好地学习观测数据的条件分布。这个模型可以称作判别模型  $D$ ，它的输入是数据空间内的任意一张图像  $x$ ，输出是一个概率值，表示这张图像属于真实数据的概率。对于生成模型  $G$  来说，它的输入是一个随机变量  $z$ ， $z$  服从某种分布，输出是一张图像  $G(z)$ ，如果它生成的图像经过模型  $D$  后的概率值很高，就说明生成模型已经比较好地掌握了数据的分布模式，可以产生符合要求的样本；反之则没有达到要求，还需要继续训练。

两个模型的目标如下所示。

1. 判别模型的目标是最大化这个公式： $E_x[D(x)]$ ，也就是甄别出哪些图是真实数据分布中的。
2. 生成模型的目标是最大化这个公式： $E_z[D(G(z))]$ ，也就是让自己生成的图被判别模型判断为来自真实数据分布。

看上去两个模型目标联系并不大，下面就要增加两个模型的联系，如果生成模型生成的图像和真实的图像有区别，判别模型要给它判定比较低的概率。这里可以举个形象的例子， $x$  好比是一种商品， $D$  是商品的检验方，负责检验商品是否是正品； $G$  是一家山寨公司，希望根据拿到手的一批产品  $x$  研究出生产山寨商品  $x$  的方式。对于  $D$  来说，不管  $G$  生产出来的商品多像正品，都应该被判定为赝品，更何况一开始  $G$  的技术水平不高，生产出来的产品必然是漏洞百出，所以被判定为赝品也不算冤枉，只有不断地提高技术，才有可能迷惑检验方。

基于上面的例子，两个模型的目标就可以统一成一个充满硝烟味的目标函数。

$$\min_G \max_D V(D, G) = E_x[\log D(x)] + E_z[\log(1 - D(G(z)))]$$

上面这个公式对应的模型架构如图 10-5 所示。

对应的模型学习算法伪代码如下所示：

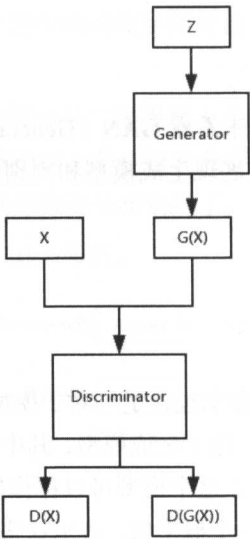


图 10-5 GAN 的基本形式

```
def GAN(G,D,X):  
    # G 表示生成模型  
    # D 表示判别模型  
    # X 表示训练数据  
    for iter in range(MAX_ITER):  
        for step in range(K):  
            x = data_sample(X)  
            z = noise_sample()  
            optimize_D(G, D, x, z)  
            z = noise_sample()  
            optimize_G(G, D, z)
```

上面的代码只是从宏观的层面介绍了模型的优化方法，其中  $K$  表示了判别模型  $D$  的迭代次数， $K$  一般大于等于 1。从上面的公式可以看出，两个模型的目标是对立的。生成模型希望最大化自己生成图像的似然，判别模型希望最大化原始数据的似然的同时，能够最小化  $G$  生成的图像的似然。既然是对立的，那么两个模型经过训练产生的能力就可能有多种情况。它们既可能上演“魔高一尺，道高一尺”，“道高一丈，魔高十丈”的竞争戏码，在竞争中共同成长，最终产生两个强大的模型；也可能产生一个强大的模型，将另一方完全压倒。

如果判别模型太过强大，那么生成模型会产生两种情况：一种情况是发现自己完全被针对，模型参数无法优化；另外一种情况是发现判别模型的一些漏洞后，它的模型

将退化，不管输入是什么样子，输出统一变成之前突破了判别模型防线的那几类结果。这种情况被称为“Mode Collapse”，有点像一个复杂强大的模型崩塌成一个简单弱小的模型，这样的模型即使优化结果很好，也不能拿去使用。

如果判别模型不够强大，它的判别不够精准，而生成模型又是按照它的判别结果生产，那么生产出的产品不会很稳定，这同样不是我们想看到的结果。

总而言之，对抗是 GAN 这个模型要面对的一个大问题。虽然论文中作者试图将两个模型共同优化的问题转换成类似 Coordinate Ascent 那样的优化问题，并证明像 Coordinate Ascent 这样的算法可以收敛，那么 GAN 这个模型也可以。不过作者在完成证明后立刻翻脸，说证明结果和实验结果不符。所以这个问题在当时也就变成了一个悬案。

### 10.2.2 GAN 的训练分析

关于 GAN 训练求解的过程，作者用了十分数学化的方式进行了推演。我们首先来证明第一步：当生成模型固定时，判别模型的最优形式。

首先将目标函数做变换：

$$\begin{aligned}\max_D V(D, G) &= E_x[\log D(x)] + E_z[\log(1 - D(G(z)))] \\ &= \int_x p_r(x) \log D(x) dx + \int_z p_g(z) \log(1 - D(G(z))) dz\end{aligned}$$

由于组成式子的两部分积分的区域不同，会对后面的计算造成困难，我们首先将两个积分区域统一。我们将生成图像  $G(z)$  的分布与真实图像  $x$  的分布做一个投射，只要判别式能够在真实数据出现的地方保证判别正确最大化即可，于是公式就变成了：

$$\begin{aligned}&= \int_x p_r(x) \log D(x) dx + \int_x p_g(x) \log(1 - D(x)) dx \\ &= \int_x [p_r(x) \log D(x) + p_g(x) \log(1 - D(x))] dx\end{aligned}$$

只要让积分内部的公式最大化，整个公式就可以实现最大化。这样问题就转变为最大化下面的公式：

$$f(D(x)) = p_r(x) \log D(x) + p_g(x) \log(1 - D(x))$$

对它进行求导取极值，可以得到：

$$f'(D(x)) = p_r(x) \frac{1}{D(x)} - p_g(x) \frac{1}{1 - D(x)}$$

令上面的式子为 0，我们可以得到结果：

$$D^*(x) = \frac{p_r(x)}{p_r(x) + p_g(x)}$$

这就是理论上判别式的预测结果，如果一张图像在真实分布中出现的概率大而在生成分布中出现的概率小，那么最优的判别模型会认为它是真实图像，反之则认为不是真实图像。如果生成模型已经达到了完美的状态，也就是说对每一幅图像都有：

$$p_r(x) = p_g(x), \text{ 那么 } D^*(x) = \frac{1}{2}$$

接下来就可以利用上面的结果，计算当生成模型达到完美状态时，损失函数的值。我们将  $D^*(x) = \frac{1}{2}$  的结果代入，可以得到：

$$\begin{aligned} &= \int_x p_r(x) \log \frac{1}{2} dx + \int_x p_g(x) \log(1 - \frac{1}{2}) dx \\ &= -\log 2 \int_x p_r(x) dx - \log 2 \int_x p_g(x) dx \\ &= -2 \log 2 \end{aligned}$$

也就是说生成模型损失函数的理论最小值为  $-2\log 2$ 。那么，一般情况下它的损失函数是什么样子呢？我们假设在某一时刻判别式经过优化已经达到最优，所以  $D^*(x) = \frac{p_r(x)}{p_r(x) + p_g(x)}$ ，我们将这个公式代入之前的公式，可以得到：

$$\begin{aligned} &= \int_x p_r(x) \log \frac{p_r(x)}{p_r(x) + p_g(x)} dx + \int_x p_g(x) \log(1 - \frac{p_r(x)}{p_r(x) + p_g(x)}) dx \\ &= \int_x p_r(x) \log \frac{p_r(x)}{p_r(x) + p_g(x)} dx + \int_x p_g(x) \log(\frac{p_g(x)}{p_r(x) + p_g(x)}) dx \\ &= \int_x p_r(x) \log(\frac{p_r(x)}{\frac{p_r(x) + p_g(x)}{2}} \times \frac{1}{2}) dx + \int_x p_g(x) \log(\frac{p_g(x)}{\frac{p_r(x) + p_g(x)}{2}} \times \frac{1}{2}) dx \\ &= \int_x p_r(x) [\log \frac{p_r(x)}{\frac{p_r(x) + p_g(x)}{2}} + \log \frac{1}{2}] dx + \int_x p_g(x) [\log \frac{p_g(x)}{\frac{p_r(x) + p_g(x)}{2}} + \log \frac{1}{2}] dx \\ &= -\log 2 + \int_x p_r(x) [\log \frac{p_r(x)}{\frac{p_r(x) + p_g(x)}{2}}] dx - \log 2 + \int_x p_g(x) [\log \frac{p_g(x)}{\frac{p_r(x) + p_g(x)}{2}}] dx \\ &= -2 \log 2 + \text{KL}(p_r(x) || \frac{p_r(x) + p_g(x)}{2}) + \text{KL}(p_g(x) || \frac{p_r(x) + p_g(x)}{2}) \end{aligned}$$

后面的两个 KL 散度的计算公式可以转化为 **Jenson-Shannon 散度**，也就是：

$$= -2 \log 2 + 2\text{JSD}(p_{\text{data}}||p_g)$$

这其实是生成模型真正的优化目标函数。在介绍 VAE 时，读者已经了解了 KL 散度，也了解了它的一些基本知识，那么这个 JS 散度又是什么？它又有什么特性和优势？从最直观的角度，读者可以发现一个 KL 散度不具备的性质——JS 散度是对称的：

$$\text{JS}(p||q) = \text{KL}(p||\frac{p+q}{2}) + \text{KL}(q||\frac{p+q}{2}) = \text{JS}(q||p)$$

对称又能带来什么好处呢？它能让散度量更准确。接下来将用一段代码展示这其中的道理。首先给出两个离散随机变量的 KL 散度和 JS 散度的计算方法：

```
import numpy as np
import math
def KL(p, q):
    # p,q为两个list，里面存着对应的取值的概率，整个list相加为1
    if 0 in q:
        raise ValueError
    return sum(_p * math.log(_p/_q) for (_p,_q) in zip(p, q) if _p != 0)

def JS(p, q):
    M = [0.5 * (_p + _q) for (_p, _q) in zip(p, q)]
    return 0.5 * (KL(p, M) + KL(q, M))
```

下面将用 3 组实验看看两个散度的计算结果。首先选定一个简单的离散分布，然后求出它的 KL 散度和 JS 散度。在此基础上，把两个分布分别做一定的调整。首先是基础的分布：

```
def exp(a, b):
    a = np.array(a, dtype=np.float32)
    b = np.array(b, dtype=np.float32)
    a /= a.sum()
    b /= b.sum()
    print a
    print b
    print KL(a,b)
    print JS(a,b)
```



```
# exp 1
exp([1,2,3,4,5],[5,4,3,2,1])

#以下为运行结果显示
[ 0.066  0.133  0.2          0.266  0.333]
[ 0.333  0.266  0.2          0.133  0.066]
0.521
0.119
```

接下来把公式中第二个分布做修改，假设这个分布中有某个值的取值非常小，就有可能增加两个分布的散度值，它的代码如下所示：

```
# exp 2
exp([1,2,3,4,5],[1e-12,4,3,2,1])
exp([1,2,3,4,5],[5,4,3,2,1e-12])

#以下为运行结果显示
[ 0.066  0.133  0.2          0.266  0.333]
[ 9.999e-14  4.000e-01  3.000e-01  2.000e-01  1.000e-01]
2.06550201846
0.0985487692551

[ 0.066  0.133  0.2          0.266  0.333]
[ 3.571e-01  2.857e-01  2.142e-01  1.428e-01  7.142e-14]
9.662
0.193
```

可以看出 KL 散度的波动比较大，而 JS 的波动相对小。

最后修改前面的分布，代码如下所示：

```
# exp 3
exp([1e-12,2,3,4,5],[5,4,3,2,1])
exp([1,2,3,4,1e-12],[5,4,3,2,1])

这回得到的结果是这样的：
[ 7.142e-14  1.428e-01  2.142e-01  2.857e-01  3.571e-01]
[ 0.333  0.266  0.2          0.133  0.0666]
0.742
```

0.193

[ 1.000e-01 2.000e-01 3.000e-01 4.000e-01 9.999e-14]

[ 0.333 0.266 0.2 0.133 0.066]

0.383

0.098

如果将第二个实验和第三个实验做对比，就可以发现 KL 散度在衡量两个分布的差异时具有很大的不对称性。如果后面的分布在某一个值上缺失，就会得到很大的散度值；但是如果前面的分布在某一个值上缺失，最终的 KL 散度并没有太大的波动。这个例子可以很清楚地看出 KL 不对称性带来的一些小问题。而 JS 具有对称性，所以第二个实验和第三个实验的 JS 散度实际上是距离相等的分布组。

从这个小例子我们可以看出，有时 KL 散度下降的程度和两个分布靠近的程度不成比例，而 JS 散度靠近的程度更令人满意，这也是 GAN 模型的一大优势。

### 10.2.3 GAN 实战

看完了前面关于 GAN 的理论分析，下面我们开始实战。在实战之前目标函数还要做一点改动。从前面的公式中可以看出这个模型和 VAE 一样都是有嵌套关系的模型，那么生成模型  $G$  要想完成前向后向的计算，要先将计算结果传递到判别模型计算损失函数，然后将梯度反向传播回来。那么不可避免地我们会遇到一个问题，如果梯度在判别模型那边消失了，生成模型岂不是没法更新了？生成模型的目标函数如下所示：

$$\min_G V(D, G) = E_z[\log(1 - D(G(z)))]$$

如果判别模型非常厉害，成功地让  $D(G(z))$  等于一个接近 0 的数字，那么这个损失函数的梯度就消失了。其实从理论上分析这个结果很正常，生成模型的梯度只能从判别模型这边传过来，这个结果接近 0 对于判别模型来说是满意的，所以它不需要更新，梯度就没有了，于是生成模型就没法训练了。所以作者又设计了新的函数目标：

$$\min_G V(D, G) = -E_z[\log(D(G(z)))]$$

这样一来梯度又有了，生成模型也可以继续训练。当然，这个目标函数也有不足的地方。

下面来看一个具体的基于深层模型的实现——DC-GAN<sup>[4]</sup>。全称是 Deep Convolution GAN。也就是用深度卷积网络进行对抗生成网络的建模。在此之前，也有一些基于

卷积神经网络的 GAN 实现，但是相对来说，DC-GAN 的最终表现与同期的模型相比更优秀，在介绍它的论文中，作者也详细介绍了模型的一些改进细节。

- 将 Pooling 层替换成带有 stride 的卷积层
- 使用 Batch Normalization
- 放弃使用全连接层
- 将卷积层的非线性部分换成 ReLU 或者 Leaky ReLU

下面将使用 DC-GAN 的模型进行实验，这个实验使用的数据集还是 MNIST。由于 Caffe 并不是十分适合构建 GAN 这样的模型，因此这里使用另外一个十分流行且简单易懂的框架——Keras 来展示 DC-GAN 的一些细节。代码来自 <https://github.com/jacobgil/keras-dcgan>。由于 Keras 的代码十分直观，这里就直接给出源码。首先是生成模型：

```
def generator_model():
    model = Sequential()
    model.add(Dense(input_dim=100, output_dim=1024))
    model.add(Activation('tanh'))
    model.add(Dense(out_dim=128*7*7))
    model.add(BatchNormalization())
    model.add(Activation('tanh'))
    model.add(Reshape((128, 7, 7), input_shape=(128*7*7,)))
    model.add(UpSampling2D(size=(2, 2)))
    model.add(Convolution2D(out_channel=64, kernel_height=5, kernel_width
                             =5, border_mode='same'))
    model.add(Activation('tanh'))
    model.add(UpSampling2D(size=(2, 2)))
    model.add(Convolution2D(out_channel=1, kernel_height=5, kernel_width
                             =5, border_mode='same'))
    model.add(Activation('tanh'))
    return model
```

这里需要说明的一点是，这个实现和论文中的描述有些不同，不过对于 MNIST 这样的小数据集，这样的模型差异不影响效果。

判别模型的结构如下所示，仔细地读一遍就可以理解，这里不再赘述。

```
def discriminator_model():
    model = Sequential()
    model.add(Convolution2D(64, 5, 5, border_mode='same', input_shape=(1,
                                                                           28, 28)))
```

```

model.add(Activation('tanh'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Convolution2D(128, 5, 5))
model.add(Activation('tanh'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Flatten())
model.add(Dense(1024))
model.add(Activation('tanh'))
model.add(Dense(1))
model.add(Activation('sigmoid'))
return model

```

完成训练后，生成模型生成的手写数字如图 10-6 所示。



图 10-6 GAN 生成的图像

除了个别数字外，大多数数字生成得和真实数据很像。将图 10-6 和图 10-2 进行对比，我们可以发现，GAN 模型生成的数字相对而言更为“清晰”，而 VAE 模型的数字略显模糊，这和两个模型的目标函数有很大的关系。另外，两个模型在训练过程中的 Loss 曲线如图 10-7 所示。

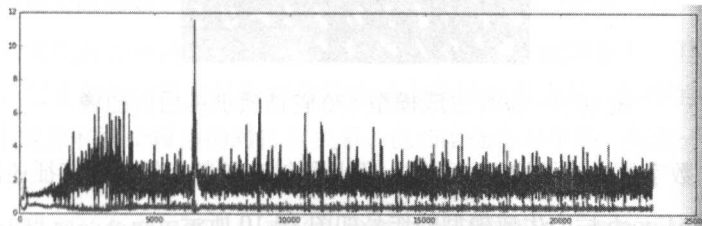


图 10-7 GAN 中生成模型和判别模型的损失函数

其中上面的曲线表示生成模型的 Loss，下面的曲线是判别模型的 Loss，虽然这两个 Loss 的绝对数值看上去不能说明什么问题，但是相信读者还是可以看出两个模型的 Loss 存在着强相关的关系，这也算是对抗过程中的此消彼长。

最终生成的数据还算令人满意，我们还很好奇，在模型优化过程中生成模型生成的图像都是什么样的呢？接下来就来观察生成图像的演变过程。在优化开始时，随机生成的图像如图 10-8 所示。

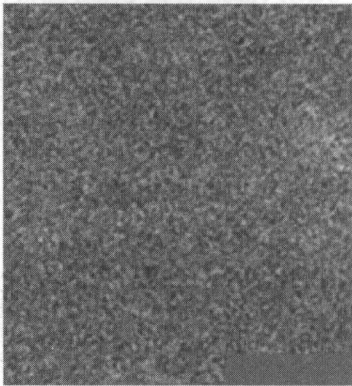


图 10-8 GAN 生成模型的初始图像

其实就是噪声图片，一点都不像数字。经过 400 轮的迭代，生成模型可以生成的图像如图 10-9 所示。

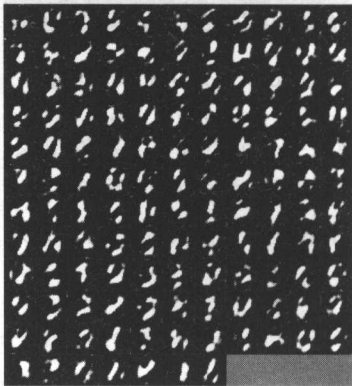


图 10-9 GAN 生成模型 400 轮迭代训练后的图像

可以看出数字的大体结构已经形成，但是能够表征数字细节的特征还没有出现。

经过 10 个 Epoch 后，生成模型的作品如图 10-10 所示。

这时有些数字已经成形，但是还有一些数字仍然存在欠缺。

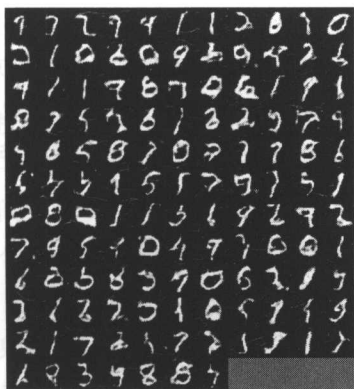


图 10-10 GAN 生成模型经过 10 个 Epoch 迭代训练后的图像

20 轮 Epoch 后的结果如图 10-11 所示。



图 10-11 GAN 生成模型经过 20 个 Epoch 迭代训练后的图像

这时的数字已经具有很强的辨识度，但与此同时，我们发现生成的数字中有大量的“1”。

当完成了所有的训练，取出生成模型在最后一轮生成的图像，如图 10-12 所示。

可以看出这里的数字质量更高一些，但是里面的“1”更多了。

从模型的训练过程中可以看出，一开始生成的数字质量都很差，但生成数字的多样性比较好，后来的数字质量比较高但数字的多样性逐渐变差，模型的特性在不断发生变化。这个现象和两个模型的对抗有关系，也和增强学习中的“探索—利用”困境很类似。

站在生成模型的角度思考，一开始生成模型会尽可能地生成各种各样形状的数字，而判别模型会识别出一些形状较差的模型，而放过一些形状较好的模型，随着学习的

进程不断推进，判别模型的能力也在不断地加强，生成模型慢慢发现有一些固定的模式比较容易通过，而其他的模式不那么容易通过，于是它就会尽可能地增大这些正确模式出现的概率，让自己的 Loss 变小。这样，一个从探索为主的模型变成了一个以利用为主的模型，因此它的数据分布已经不像刚开始那么均匀了。

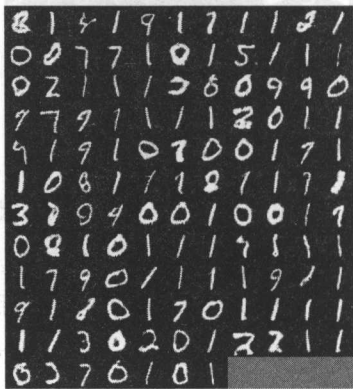


图 10-12 GAN 生成模型最终生成的图像

如果这个模型继续训练下去，生成模型有可能进一步地利用这个模式，这和机器学习中的过拟合也有很相近的地方。

### 10.3 Info-GAN

本节将要介绍 GAN 模型的一个变种——InfoGAN<sup>[5]</sup>，它要解决隐变量可解释性的问题。前面提到 GAN 的隐变量服从某种分布，但是这个分布背后的含义却不得而知。虽然经过训练的 GAN 可以生成新的图像，但是它却无法解决一个问题——生成具有某种特征的图像。例如，对于 MNIST 的数据，生成某个具体数字的图像，生成笔画较粗、方向倾斜的图像等，这时就会发现经典的 GAN 已经无法解决这样的问题，想要解决就需要想点别的办法。

首先想到的方法就是生成模型建模的方法：挑出几个隐变量，强制指定它们用来表示这些特性的属性，例如数字名称和方向。这样看上去似乎没有解决问题，但这种方法需要提前知道可以建模的隐变量内容，还要为这些隐变量设置好独立的分布假设，实际上有些麻烦又不够灵活。本节的主角——InfoGAN，将从信息论角度，尝试解决 GAN 隐变量可解释性问题。

### 10.3.1 互信息

介绍算法前要简单回顾机器学习中的信息论基本知识。第2章已经介绍了熵和“惊喜度”这些概念，熵衡量了一个随机变量带来的“惊喜度”。本节要介绍的概念叫做互信息，它衡量了随机变量之间的关联关系。假设随机事件 A 的结果已经知道，现在要猜测某个事件 B 的结果，那么知道 A 的取值对猜测 B 有多大帮助？这就是互信息要表达的东西。

我们以掷骰子为例，如果我们知道手中的骰子是不是“韦小宝特制”骰子这件事，那么它会对我们猜测最终投掷的点数有帮助吗？当然有帮助，因为一旦确定这个骰子是“韦小宝特制”，那么骰子点数是几这个信息就变得没有“惊喜”了。同理，“美国第45届总统是谁”这个消息对我们手中骰子投掷出的点数这个事情就没那么多帮助了，所以这两件事情的互信息就低，甚至可以说这两个事件是相互独立的。

了解了上面比较直观的例子，下面就可以给出连续随机变量  $X, Y$  互信息的计算公式：

$$I(X; Y) = \int_{x \in X} \int_{y \in Y} P(x, y) \log \frac{P(x, y)}{P(x)P(y)} dx dy$$

上面的公式可以做如下变换：

$$\begin{aligned} I(X; Y) &= \int_{x \in X} \int_{y \in Y} P(x, y) \log \frac{P(x, y)}{P(x)P(y)} dx dy \\ &= \int_{x \in X} \int_{y \in Y} P(x, y) \log \frac{P(x, y)}{P(x)} dx dy - \int_{x \in X} \int_{y \in Y} P(x, y) \log P(y) dx dy \\ &= \int_{x \in X} \int_{y \in Y} P(x)P(y|x) \log P(y|x) dx dy - \int_{y \in Y} \log P(y) \int_{x \in X} P(x, y) dx dy \\ &= \int_{x \in X} P(x) \int_{y \in Y} P(y|x) \log P(y|x) dx dy - \int_{y \in Y} \log P(y) P(y) dx dy \\ &= - \int_{x \in X} P(X) H(Y|x) dx + H(Y) \\ &= H(Y) - H(Y|X) \end{aligned}$$

就可以发现互信息的进一步解释：它可以变为熵和条件熵的差。同样地，这个公式还可以转变为：

$$I(X; Y) = H(X) - H(X|Y)$$

最终表示为熵和条件熵的差距。用通俗的话解释，两个随机变量的互信息就是在知道和不知道一个随机变量取值的情况下，另一个随机变量“惊喜度”的变化。互信息



的计算方法的代码如下所示：

```
import numpy as np
import math

def mutual_info(x_var, y_var):
    sum = 0.0
    x_set = set(x_var)
    y_set = set(y_var)
    for x_val in x_set:
        px = float(np.sum(x_var == x_val)) / x_var.size
        x_idx = np.where(x_var == x_val)[0]
        for y_val in y_set:
            py = float(np.sum(y_var == y_val)) / y_var.size
            y_idx = np.where(y_var == y_val)[0]
            pxy = float(np.intersect1d(x_idx, y_idx).size) / x_var.size
            if pxy > 0.0:
                sum += pxy * math.log((pxy / (px * py)), 10)
    return sum
```

下面随意给出一对随机变量和它们的概率分布，并用上面的代码分析这对变量的互信息：

```
a = np.array([0,0,5,6,0,4,4,3,1,2])
b = np.array([3,4,5,5,3,7,7,6,5,1])
print mutual_info(a,b)
# 0.653
```

```
a = np.array([0,0,5,6,0,4,4,3,1,2])
b = np.array([3,3,5,6,3,7,7,9,4,8])
print mutual_info(a,b)
# 0.796
```

很明显，下面一组数据的相关性更强，知道其中一个随机变量的取值，就会非常容易猜出同一时刻另外一个随机变量的采样值。如果我们进一步观察第二组数据，会发现任意一组数据的熵都是 0.796，也就是说当知道其中一个随机变量的值后，它们的条件熵就变成了 0，另一个随机变量变得完全“惊喜”了。虽然条件熵为 0 这个信息并没有展现在互信息的数值中，但互信息实际上就是在衡量一个相对的信息差距，并不像熵那样衡量信息绝对量。

其实数学包含了很多人生哲理和智慧。人的一生实际上一直在和熵作斗争，每个人的生活轨迹的熵意味着什么？一个人未来的不确定性？一个人未来的“惊喜”程度？有的人说自己“一辈子也就这样了”的时候，是不是表示这个人的未来已经从一个随机变量变成了常量，它的熵变成了0？为什么人们总是向往青春，是不是因为那些年华充满了各种不确定性与精彩，可以理解为熵很大？

“身体和灵魂，总有一个在路上”，是不是标榜追求最大熵的一个口号？“公务员这种稳定工作才是好工作”是不是一种追求最小化熵的行为呢？那么对于一个人来说，究竟是熵越大越好，还是熵越小越好？

回到问题，互信息在这个问题中有什么用？如果说隐变量的确定对确定生成图像的样子有帮助，那么隐变量和最终的图像之间的互信息就应该很大：当知道了隐变量这个信息，图像的信息对变得更确定了。所以 InfoGAN 这个算法就是要通过约束互信息使隐变量“更有价值”。

### 10.3.2 InfoGAN 模型

那么，InfoGAN 模型的具体形式是什么样的呢？如果把互信息定义为损失函数的一部分，这部分损失函数就是 InfoGAN 中基于经典 GAN 修改的部分。前面的小节已经推导出了互信息的公式，那么在具体计算时要使用哪个公式计算呢？

- $I(X; Z) = H(X) - H(X|Z)$
- $I(X; Z) = H(Z) - H(Z|X)$

最终的选择是后者，因为图像  $X$  的分布太难确定，求解它的熵肯定相当困难，所以前者的第一项非常难计算。当然，即使选择了第二项，这个公式也不是很好优化，因为其中还有一个后验项  $P(Z|X)$  需要求解，这也是个大麻烦，不过这里可以使用本书多次提到的方法——Variational Inference 求解这个后验项。

在介绍 VAE 时我们曾经运用过 Reparameterization Trick 这个方法，这里将再次采用类似的方法。在 VAE 中，Trick 公式是  $z^{(i)} = g_{\phi}(X + \epsilon^{(i)})$ ，在 Encoder 的过程中，输入部分被分解成确定部分和不确定部分，然后利用一个高维非线性模型拟合输入到输出的映射。这里要求出的  $X$ ，和 VAE 正好相反，需要的是这样的一个公式：

$$X = g_{\phi}([c, z] + \epsilon)$$

其中  $c$  表示与图像有相关关系的隐变量， $z$  表示与图像无关的隐变量。于是互信息计算公式就变成了：

$$= E_{c \sim P(c), \epsilon} [\log Q(c|g_{\phi}([c, z] + \epsilon))] + H(c)$$

从实践上讲， $\epsilon$  项可以忽略，于是公式可以做进一步简化：

$$= E_{c \sim P(c)} [\log Q(c|g_\phi([c, z]))] + H(c)$$

接下来将期望用蒙特卡罗方法代替，训练时可以通过计算大量样本求平均来代替期望值，于是公式又变成了：

$$= \frac{1}{N} \sum_{c \sim P(c)} [P(c) \log Q(c|g_\phi([c, z])) + P(c) \log P(c)]$$

这个方程变简单了很多。当然，我们也看出上面的公式中我们有一个  $Q$ ，这个  $Q$  函数可以理解为一个 Encoder，这部分模型在经典 GAN 中并不存在，但是在实际建模过程中，由于 Encoder 和判别模型的输入相同，且模型目标比较相近，因此二者部分网络结构可以共享。论文的作者提供了 InfoGAN 的源码，代码的链接在 <https://github.com/openai/InfoGAN>，代码使用的框架为 TensorFlow，感兴趣的读者可以自行阅读。模型实现的结构如图 10-13 所示。

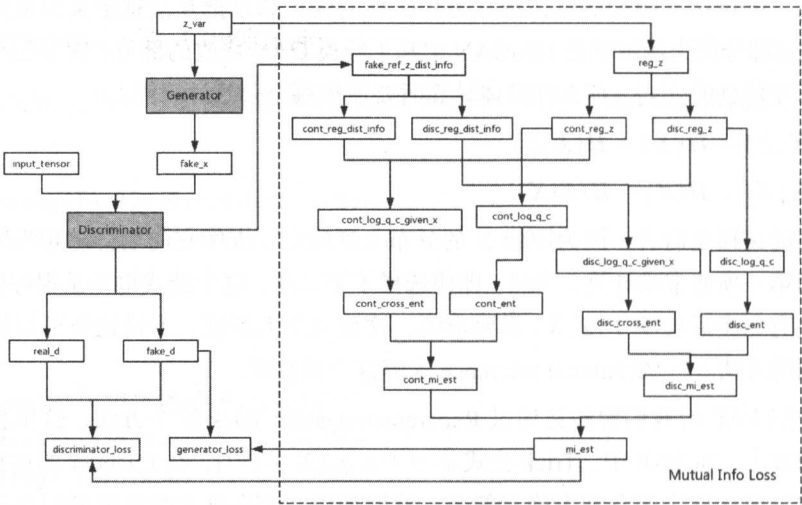


图 10-13 InfoGAN 模型结构图

虚线部分表示的就是计算互信息的目标函数，这部分内容看似比较复杂，实则不然。由于 InfoGAN 模型中定义了两种类型的随机变量——服从 Categorical 分布、用于表示数字内容的离散类型变量，和服从均匀分布用于表示其他连续特征的连续型变量，而两种类型的变量在计算熵的方法不同，因此上面的计算图对它们进行分情况处理。

互信息计算起始于如下两个变量。

- *reg\_z*: 表示了模型开始随机生成的隐变量。
- *fake\_ref\_z\_dist\_info*: 表示了经过 Encoder 计算后的隐变量分布信息。

接下来，根据连续型和离散型的分类，两个变量分成了以下四个变量。

- *cont\_reg\_z*: *reg\_z* 的连续变量部分
- *cont\_reg\_dist\_info*: *fake\_ref\_z\_dist\_info* 的连续变量部分
- *disc\_reg\_z*: *reg\_z* 的离散变量部分
- *disc\_reg\_dist\_info*: *fake\_ref\_z\_dist\_info* 的连续变量部分

接下来，四个变量两两组队完成了后验公式  $P(c) \log Q(c|g_\phi([c, z]))$  的计算：

- *cont\_log\_q\_c\_given\_x*: 连续变量的后验
- *disc\_log\_q\_c\_given\_x*: 离散变量的后验

同时，输入的隐变量也各自完成先验  $P(c) \log P(c)$  的计算：

- *cont\_log\_q\_c*: 连续变量的先验
- *disc\_log\_q\_c*: 离散变量的后验

由于上面的运算全部是元素级的计算，还要把向量求出的内容汇总，得到  $\sum P(c) \log Q(c|g_\phi([c, z]))$  和  $\sum P(c) \log P(c)$ 。

- *cont\_cross\_ent*: 连续变量的交叉熵
- *cont\_ent*: 连续变量的熵
- *disc\_cross\_ent*: 离散变量的交叉熵
- *disc\_ent*: 离散变量的熵

接下来，根据互信息公式两两相减，得到各自的互信息损失。

- *cont\_mi\_est*: 连续变量的互信息
- *disc\_mi\_est*: 离散变量的互信息

最后将两者相加就得到了最终的互信息损失。

模型在训练前定义了 12 个和图像有强烈互信息的随机变量，其中 10 个变量表示显示的数字，它们组成一个 Categorical 的离散随机向量；另外 2 个是服从范围为  $[-1, 1]$  的连续随机变量。训练完成后，调整离散随机变量输入并生成图像，得到如图 10-14 所示的数字图像。

可以看出模型很好地识别了这些数字。调整另外两个连续随机变量，可以生成如图 10-15 所示的数字图像。

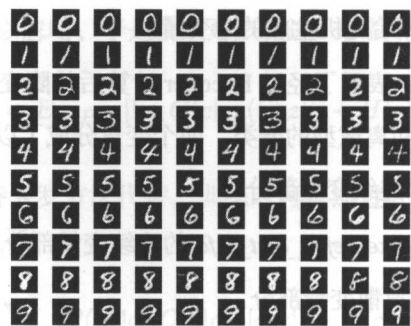


图 10-14 10 个离散随机变量对生成数字的影响

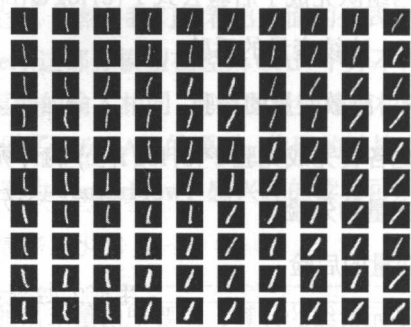


图 10-15 2 个连续随机变量对生成数字的影响

可以看出，这两个连续随机变量学到了数字粗细和倾斜的特征，而且这是在完全没有暗示的情况下完成的。可见 InfoGAN 模型的能力。

到此 InfoGAN 的介绍就结束了。从这个模型可以看出，在经典 GAN 模型基础上添加更多的内容会产生更多意想不到的效果。

## 10.4 Wasserstein GAN

前面章节介绍了 GAN 的基本概念，也介绍了一些有关 GAN 的理论性质。可以看出，GAN 的理论是比较脆弱的（目前深度学习的理论普遍比较弱），而且在训练 GAN 时经常会遇到一些问题，例如训练过程中梯度消失、优化过程不稳定、Loss 数值在展现优化效果方面较弱。这些问题归根结底还是在于模型本身的优势，也是模型的劣势——对抗。前面提到过“魔高一尺，道高一丈”的状态实际上是理想状态，攻守双方一旦失衡，模型的效果就会大打折扣，模型优化的难度也将大大增加。为了解决这些问题，很多专家展开了进一步的研究，试图解决训练过程中的这些问题。如果说训练 CNN 模型

是“炼丹”的话，那么训练 GAN 模型就是“炼丹中的战斗机”。Wasserstein GAN 模型的出现为大家从 Optimal Transport 的角度揭示了 GAN 模型背后更多的理论，也为大家带来了更稳定的 GAN 训练方法。

### 10.4.1 分布的重叠度

模型优化的难度如何增大呢？可以想象，GAN 模型的训练数据往往是某一个类别的图像数据，在高维空间下，它们的分布显然不能覆盖整个空间。因此，真实数据  $P_r$  在浩瀚的图像空间中显得十分渺小；同理，GAN 模型描绘的数据分布  $P_g$  也不算大，于是两个分布好比茫茫人海中的你我，稍不留神就会错过。

为了证明这个问题，文章 *Towards Principled Methods for Training Generative Adversarial Networks*<sup>[6]</sup> 给出了一个完整的逻辑：真实数据和生成数据分布的支撑面在完整的图像空间中十分渺小，它们的测度可以忽略不计。因此两个分布就有很大概率并不相交。只要分布不相交，判别模型就有很大的机会将两者完美分开，这样根据目标函数，生成模型将不再获得任何梯度，模型的梯度消失，优化也被停止了。

首先简单证明第一个结论：真实数据和 GAN 生成的数据在图像空间出现的概率很低，概率分布的测度基本可以认为是 0。GAN 的生成模型是用 CNN 模型的模块搭建而成的。那么它的主要组成部分就是卷积层、全连接层、非线性部分和 Pooling 层等，这里称这个函数为  $g: Z \rightarrow X$ 。实际上非线性层的函数都可以想象成某种线性变换，所以 GAN 模型整体可以看作是某种复杂的线性变换。

这里假设 GAN 模型使用的是 ReLU 或者 Leaky ReLU 非线性函数，那么在非线性部分的计算中，模型会进行类似投影操作一样的线性变换。暂时不考虑 Pooling 这样改变维度的层次，并且把卷积层和全连接层统一成全连接计算形式，那么 GAN 模型可以表示为：

$$g(z) = D_n W_n \cdots D_1 W_1 z$$

其中  $W$  代表线性层或者全连接层， $D$  代表非线性层。很容易可以看出，GAN 中隐变量的维度直接决定了输出图像分布的流型的维度，线性变换无法无限度地扩大原始数据的维度。如果输入的隐变量是 100 维，实际的图像空间为 1000 维，通过生成模型的计算，最终生成的所有的图像并不会覆盖整个 1000 维的图像。那么根据测度的性质，低维点集在高维空间下的测度为 0，所以说明图像分布在完整的图像空间中仍属极为罕见。既然真实分布与生成分布都十分罕见，那么它们相遇的概率也就微乎其微了。

其次是第二个结论：如果两个数据分布不相交，那么存在一个完美分离二者的判别模型。如果两个分布  $P_r$  和  $P_g$  各自的支撑面（概率不为 0 的空间子集）不相交，那么我

们就可以找到一个最优且光滑的判别映射  $D^* : \chi \rightarrow [0, 1]$ 。这个映射对真假数据判别的精确率为 100%，且在两个分布的支撑面区域，梯度  $\nabla_x D^*(x) = 0$ 。这个定理是一个和拓扑学联系紧密的定理，它的证明过程用到了大名鼎鼎的乌雷松分离引理（Urysohn's Lemma）。

设  $X$  是一个拓扑空间， $[a, b]$  是一个闭区间，则  $X$  是一个正规空间（normal space）当且仅当对于  $X$  中任意两个无交的闭集  $A$  和  $B$ ，存在一个连续映射  $f : X \rightarrow [a, b]$  使得当  $x \in A$  时  $f(x) = a$ ， $x \in B$  时  $f(x) = b$ 。

结论的证明关键在于如何把乌雷松引理套在这个问题上。如果将上面定理中的  $A$  换成  $P_g$ ， $B$  换成  $P_r$ ，同时将  $a$  换成 0， $b$  换成 1，那么定理就可以帮助证明，完美分离两个分布的连续映射是存在的。这虽然不代表判别模型一定可以优化成这样的最优函数，但还是说明了 GAN 训练时梯度消失的潜在危险。

实际上完全不相交的情况虽然常见，但是它并不能包含所有的情况。为了考虑更多的情况，证明的场景需要扩展到更多的情况下。如果两个空间有了一点接触，是不是判别模型就没有办法破坏训练了呢？并不是。下面的推演将两个空间的关系放宽，两个空间的关系可以是任意关系，但是接下来的证明同样残酷——即使两个空间相交，它们相交的方式也分两种：一种情况下，两个空间有明显的重叠区域；另一种情况下，两个空间只是有很少量的相交。这就像在茫茫人海中，有的人从相遇相知，最终形成了很好的关系；有的却只有一面之缘。

对于关系很好的情况，判别模型想把它们完全分开就显得十分困难，只要无法完全分开，梯度就会存在，优化就会继续；但是对于“一面之缘”的情况，判别模型还是可以做到几乎完美地分离，这样即使模型的梯度存在，它的数值也会很小，那么优化需要的时间就会很长。所谓“前世的 500 次回眸才换来今生的擦肩而过”，擦肩而过没有注意到对方？不好意思，再来 500 次吧。问题在于模型训练能够等待这样的浪漫故事再继续吗？当然不能。

根据上面的分析，我们当然希望两个分布的支撑面相交，是大面积、有质量的相交，而不是简单的相交。论文中给出了对于“简单相交”（Intersect Transversally）的精确定义：如果两个流型  $M$  和  $P$  相交，且对于任意一个交点  $x = M \cap P$ ，这个点在  $M$  上的切空间和  $P$  上的切空间之和等于外围空间的切平面，那么就认为两个分布所在的流型是“泛泛之交”。比方说，三维空间下有两个流形，它们的相交区域是一条线，这样的相交就可以认为是“简单相交”。“简单相交”的场景如图 10-16 所示。图中展示了两个流形  $A$ 、 $B$ ，其中的相交的区域用点虚线表示，可以看出，相交的区域和不相交的区域相比要小太多。

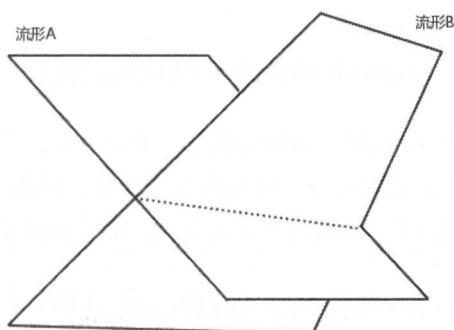


图 10-16 “简单相交”的示意图

另外一个概念是 **Perfectly Align**，只要存在一个  $x = M \cap P$ ，使得两个流型不在这一点“泛泛之交”，而是很深入地交织在一起，就认为它们是 **Perfectly Align**。

这两种相交会有多大的差别呢？在前面我们聊过 KL 散度，KL 散度能够成为一个有意义的度量函数的前提条件是待测的两个分布的支撑面尽可能地重合，如果不重合，测量分布的差距就变得没有意义，数值上的表现也会很差。下面的一个引理就来揭示这个世界残忍的一面：**Perfectly Align** 是几乎不可能的。

对于正规空间的两个维度远小于完成空间的流型  $M$  和  $P$ ，令  $\eta$  和  $\eta'$  是两个相互独立的连续随机变量，我们定义经过平移操作的流型  $\tilde{M} = M + \eta$ ， $\tilde{P} = P + \eta'$ ，那么两个经过平移的流型不会 **Perfectly Align** 的概率为 1。

经过上面的层层推演，可以看出两个分布基本上很难相交，尤其是在训练初期；一旦判别模型抓住了机会，得到了充分训练，生成模型的优化就可能变得艰难。

#### 10.4.2 两种目标函数存在的问题

在 GAN 的论文介绍中，作者提出了两种优化模型的方式，而实际上这两种训练方式都存在一些问题。

首先是第一种目标函数：

$$L(D, G) = E_{x \sim P_r} [\log D(x)] + E_{z \sim P_z} [\log(1 - D(G(z)))]$$

在介绍 GAN 的论文中，作者已经介绍过，这种目标函数在训练过程中会产生梯度消失的现象。10.2 节曾进行推导，当判别模型经过充分优化后，生成模型的损失函数将



变为：

$$L(G) = -2 \log 2 + 2 \text{JSD}(p_{\text{data}} || p_g)$$

可见优化的关键在于 JS 散度。如果按照上一节的分析，当两个分布没有完成 Perfectly Align，且判别模型完美训练时，JS 散度会变成什么样呢？由于  $P_r$  和  $P_g$  不相交，所以  $\text{KL}(P_r || P_g)$  和  $\text{KL}(P_g || P_r)$  的值均为正无穷大，而它们的 JS 散度为：

$$\begin{aligned} \text{JSD}(P_r || P_g) &= \frac{1}{2} \text{KL}(P_r || P_{\text{mean}}) + \frac{1}{2} \text{KL}(P_g || P_{\text{mean}}) \\ &= \frac{1}{2} \left[ \int_{x \in M} P_r(x) \log \frac{P_r(x)}{P_{\text{mean}}(x)} dx + \int_{x \in P} P_g(x) \log \frac{P_g(x)}{P_{\text{mean}}(x)} dx \right] \\ &= \frac{1}{2} \left[ \int_{x \in M} P_r(x) \log \frac{P_r(x)}{\frac{1}{2} P_r(x)} dx + \int_{x \in P} P_g(x) \log \frac{P_g(x)}{\frac{1}{2} P_g(x)} dx \right] \\ &= \frac{1}{2} \left[ \int_{x \in M} P_r(x) \log 2 dx + \int_{x \in P} P_g(x) \log 2 dx \right] \\ &= \log 2 \end{aligned}$$

那么按照上面的推导，这个目标函数在绝大多数情况下都是 0，这也就是 GAN 的目标函数在一些情况下难以收敛的原因。可以看出，判别式模型越接近最优结果，梯度值也越小：

$$\lim_{||D-D^*|| \rightarrow 0} \nabla_{\theta} E_{z \sim p(z)} [\log(1 - D(g_{\theta}(z)))] = 0$$

在介绍 GAN 的论文中，作者也发觉了上面这种目标函数的问题，于是提出了另外一种优化方式：

$$\Delta \theta = \nabla_{\theta} E_{z \sim p(z)} [-\log D(g_{\theta}(z))]$$

这种目标函数可以避免梯度消失，但是经过推导分析，发现它的公式形式实际上充满了矛盾。当判别模型达到最优时，目标函数变为：

$$E_{z \sim p(z)} [-\nabla_{\theta} \log D^*(g_{\theta}(z)) |_{\theta=\theta_0}] = \nabla_{\theta} [\text{KL}(P_{g_{\theta}} || P_r) - 2 \text{JSD}(P_{g_{\theta}} || P_r)] |_{\theta=\theta_0}$$

可以看出，为了使目标函数最小化，等式右边的第一项要尽可能地小，第二项要尽可能地大，为了实现这个效果，函数可以有两种选择。

1. 让两个项同时变小，大家都变小了，最终结果也就变小了。这条路是正路。
2. 利用 KL 散度的不对称性，让两者都变大，但是控制 KL 散度变大的幅度，让 JS 散度尽量变大。由于 KL 散度具有不对称的性质，完成这个任务是可能的。这条路显然是歪路，而且这条路产生的结果就是实验中有时会遇到的“Mode Dropping”。

从上面的分析来看，一旦判别模型接近最优，无论哪种目标函数都会遇到难以收敛的困难，因此寻找全新的优化目标成了十分重要的事情。

10.4.3 Wasserstein 距离

既然上面提到的两种目标函数都存着一些问题，那我们就从源头解决问题，修改目标函数，作者引出了本节的主角——Wasserstein 距离：

$$W(P,Q)=\inf_{\gamma\in\Gamma}\int_{\mathcal{X}\times\mathcal{X}}||x-y||_2d\gamma(x,y)$$

这个公式比较抽象，如果想理解其中的含义，需要对实变函数、泛函分析、测度论等数学学科有一定的了解。这里先不直接解释它，我们从一个简单且概念相近的例子入手，介绍例子中的计算公式，然后慢慢引导到 Wasserstein 距离这个公式上来。

这个例子是最优运输问题的一个简化版问题。假设我国某省某市要修建一座立交桥，修立交桥需要一定数量的石块，城市附近正好有三个采石点，我们要将采石点的石头运到指定的三个桥梁建造地点，它们的关系如图 10-17 所示。

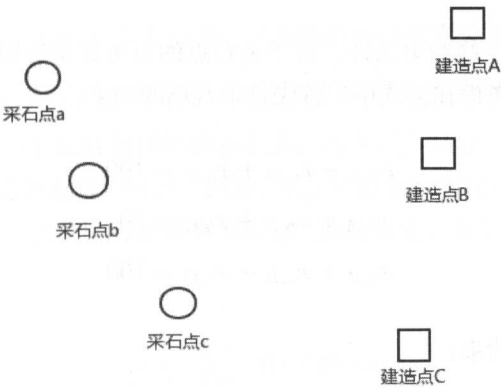


图 10-17 采石点与建造点的示意图

我们假设运输石头的代价只与运输的石头数量和运输距离有关。市委书记说了，一定要节约、减小花费，于是我们提前调研了采石点的石块储量和建造点的石头需求量。

- 采石点 a 处有 100 单位的石头
- 采石点 b 处有 50 单位的石头
- 采石点 c 处有 100 单位的石头
- 建造点 A 处需要 100 单位的石头

- 建造点  $B$  处需要 50 单位的石头
- 建造点  $C$  处需要 100 单位的石头

同时，采石点与建造点的距离如表 10-1 所示。

表 10-1 采石点与建造点的距离

	A	B	C
a	10	20	30
b	25	25	25
c	20	30	10

假设每搬运 1 单位石头行走 1 单位距离要花费 1 元，那么最省钱的搬运路径是什么呢？

这个问题的建模其实并不复杂。令  $x_{s,t}$  表示从采石点  $s$  处到建造点  $t$  处运输的石头数量，那么这个问题的目标函数就可以用下面的公式表示：

$$\min(10x_{a,A} + 20x_{a,B} + 30x_{a,C} + 25x_{b,A} + 25x_{b,B} + 25x_{b,C} + 20x_{c,A} + 30x_{c,B} + 10x_{c,C})$$

除此之外，还有一些约束条件。由于采石点的石头数量有限，建造点的需求量有限，这些约束条件要考虑在公式中。首先是采石场的约束：

$$x_{a,A} + x_{a,B} + x_{a,C} = 100$$

$$x_{b,A} + x_{b,B} + x_{b,C} = 50$$

$$x_{c,A} + x_{c,B} + x_{c,C} = 100$$

其次是建造点的约束：

$$x_{a,A} + x_{b,A} + x_{c,A} = 100$$

$$x_{a,B} + x_{b,B} + x_{c,B} = 50$$

$$x_{a,C} + x_{b,C} + x_{c,C} = 100$$

有了这些约束条件，再加上目标函数，我们就可以求解出最优的运输方案和最少的花费。关于这个线性规划问题的求解方法不是本书的重点，这里就不再赘述了。

到目前为止，这个例子还看不出任何和 Wasserstein 距离有关的线索，可能读者都有些忘记了 Wasserstein 的公式形式了。下面我们将这个问题的公式进行一定的归纳，这

样两者的关系就被拉近了。为了拉近两者的距离，这里需要再定义三个变量：用  $d_{s,t}$  表示从采石点  $s$  到建造点  $t$  的距离，用  $m_s(s)$  表示采石点  $s$  处的石头量，用  $m_t(t)$  表示建造点  $t$  处的石头量。 $\pi(s,t)$  表示石头的移动方案集合，每一个元素表示从  $s$  到  $t$  的移动方案，那么上面的所有公式就可以浓缩成下面的样子：

$$\begin{aligned} \min \sum_{\pi(s,t)} d_{s,t} x_{s,t} \\ s.t. \\ \sum_t x_{s,t} = m_s(s), \forall s \\ \sum_s x_{s,t} = m_t(t), \forall t \end{aligned}$$

现在看上去两者的相似度高了不少，为了让它们更相近，我们将采石场的石块供应量和建造场的需求量做归一化，也就是让每一个数字除以各自组合中的总和。比方说采石场  $a$ ，现在它拥有 0.4 的石块，建造场  $A$  也只需要 0.4 的石块。这种变化不会影响最终的结果，但是它却让  $m_s(s)$  和  $m_t(t)$  拥有了概率分布函数的性质——函数值为正，总和为 1。为了迎接这种变化，我们将符号做一定的改变。

- $m_s(s)$  变成  $\mu(s)$ ，表示石块来自某一采石场的离散概率分布
- $m_t(t)$  变成  $v(t)$ ，表示石块去往某一建造场的离散概率分布

可以想象，如果一个采石场提供的石头多，那么石头从这个采石场运出的概率也会相应增加，同理建造处也是一样的。同理， $x_{s,t}$  可以表示石块从某一采石场到某一建造场的离散随机变量，它服从的分布为  $\gamma(s,t)$ 。到此为止，我们将问题再次描述一遍，它就变成了这个样子：

$$\begin{aligned} \min_{\gamma(s,t)} \sum_{(s,t) \in \Gamma} \gamma(s,t) d(s,t) \\ s.t. \\ \sum_t \gamma(s,t) = \mu(s), \forall s \\ \sum_s \gamma(s,t) = v(t), \forall t \end{aligned}$$

到此为止，升级版的问题介绍完了，还差最后一点不同，Wasserstein 距离是用积分的形式计算的，而这个例子是用求和的形式计算的。于是我们将例子中的几个采石场与建造点想象成一片连续的采石场与一片连续的建造点，如图 10-18 所示。

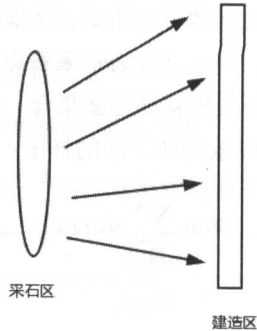


图 10-18 连续的采石场与建造点示意图

同时从上面的例子中可以发现， $\mu(s)$  和  $\nu(t)$  是  $\gamma(s, t)$  的边缘分布，于是整个问题可以转换成：

$$\begin{aligned} &\min_{\gamma(s,t)} \int_{\Gamma} d(s,t) d\gamma(s,t) \\ \text{s.t. } &\gamma(s,t) \text{ 要满足边缘分布的约束} \end{aligned}$$

到这里问题的描述和 Wasserstein 距离几乎一致了。这其中还相差一些抽象的概念。这里再回顾一下 Wasserstein 距离的公式：

$$W(P, Q) = \inf_{\gamma \in \Gamma} \int_{\mathcal{X} \times \mathcal{X}} \|x - y\|_2 d\gamma(x, y)$$

首先定义两个概率分布  $P$  和  $Q$  所在的空间  $\mathcal{X}$ ，并定义一个积空间  $\mathcal{X} \times \mathcal{X}$ ，空间的每一个元素表示了一个点对  $(x, y)$ ，其中  $x$  表示  $P$  支撑域中的点， $y$  表示  $Q$  的点，所以这个点对表示了两个分布的点对移动方案——把  $x$  移动到  $y$ 。同时  $\gamma(x, y)$  表示了这个积空间下的概率分布，也是对  $P$ 、 $Q$  两个空间下的移动量进行定量分配的函数，移动分配的数量和概率值有关。由于  $P$ 、 $Q$  两个分布本身对于每一点拥有概率值，相当于运输石头问题中的约束，所以  $\gamma(x, y)$  的分布形式也要有一定的约束。当然，满足这些约束的分布非常多，如果把这些满足约束的分布组成一个集合，经过证明可以发现，这个集合中的元素可以构成一个空间，它是用这个点的移动量表述两个分布的距离所有分布函数组成的积空间中的一个子空间。Wasserstein 距离就是要从这个空间中找到移动量最少的那个点，这个点就代表了最优移动策略，也就是那个联合概率分布。

上述概率和理论描述还存在一些距离，比方说没有提到测度论相关的一些概念，不过从表述上看更容易理解，这样即使不去接触复杂的理论，也能明白距离的含义。

### 10.4.4 Wasserstein 距离的优势

论文 *WassersteinGAN*<sup>[7]</sup> 中，作者还举了一个实际的例子，用来说明 Wasserstein 距离相比其他度量函数的优势。这个例子主要展现了计算两个支撑面几乎不重合的函数的距离时，几种度量函数的表现。进行比较的度量函数共有四个。

1. KL 散度：

$$\text{KL}(P, Q) = \int_{x \in A} P(x) \log \frac{P(x)}{Q(x)} dx$$

2. JS 散度函数：

$$\text{JS}(P, Q) = \frac{1}{2} [\text{KL}(P || \text{mean}) + \text{KL}(Q || \text{mean})]$$

3. 全变分 (Total Variation) 度量函数：

$$\delta(P_r, P_g) = \sup_{A \in \Sigma} |P_r(A) - P_g(A)|$$

4. Wasserstein 度量函数：

$$W(P, Q) = \inf_{\gamma \in \Gamma} \int_{\mathcal{X} \times \mathcal{X}} \|x - y\|_2 d\gamma(x, y)$$

公式中的  $A$  可以理解成它们的支撑面的并集， $\sup$  表示上界， $\inf$  表示下界。

全变分函数的公式看上去比较难以计算，但是幸运的是，在有限的概率空间下，全变分度量的计算可以转化为 L1 范数的计算：

$$\delta(P_r, P_g) = \frac{1}{2} \|P - Q\|_1 = \frac{1}{2} \sum_{\omega \in \Omega} |P(\omega) - Q(\omega)|$$

待计算的两个分布如下所示：

1. 令  $Z$  为  $[0, 1]$  之间的均匀分布，那么分布  $P_0$  为  $(0, Z)$  上的均匀分布。
2. 令  $\theta$  服从实数上的任意一种分布，那么分布  $P_\theta$  为  $(\theta, Z)$  上的均匀分布。

下面就来求解使用 4 种度量函数的情况下这两个分布的距离。

首先是 KL 散度，如果两个分布存在不相交的支撑集，也就是说  $\theta \neq 0$ ，那么很显然根据公式，KL 散度的值为  $+\infty$  如果； $\theta = 0$ ，那么也很显然，KL 散度的值为 0。

接下来是 JS 散度，其实在 10.4.2 节中读者已经知道了 JS 散度的表现，当  $\theta \neq 0$  时，根据公式 JS 散度的值为  $\log 2$ ，如果  $\theta = 0$ ，JS 散度的值为 0。

然后是全变分距离，根据上面给出的公式可以推出：

$$\delta(P_0, P_\theta) = \frac{1}{2} \int_{x \in \text{support}(P_0) \cup \text{support}(P_\theta)} |P_0(x) - P_\theta(x)| dx$$

当两个分布处于同一支撑面时，因为两者的分布形式相同，所以结果为 0，如果支撑面不同，就有：

$$= \frac{1}{2} \left[ \int_{x \in \text{support}(P_0)} |P_0(x)| dx + \int_{x \in \text{support}(P_\theta)} |P_\theta(x)| dx \right] = 1$$

最后是 Wasserstein 距离，从定义中可以得出，如果把两个分布想象成 2 维空间中的两根长度为 1 的线，那么它们的距离就是把一条线平移到另一条线的过程中经过区域的面积，如图 10-19 所示。

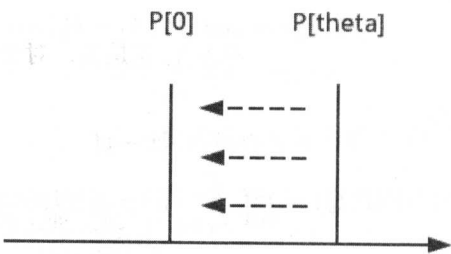


图 10-19 Wasserstein 距离计算演示

因此 Wasserstein 距离不需要分情况讨论，于是有：

$$W(P_0, P_\theta) = \int_0^{|\theta|} 1 dx = |\theta|$$

将 4 种度量的结果总结起来，可以得到表 10-2 所示的结果。

表 10-2 四种度量的计算结果

	KL 散度	JS 散度	Total Variation 距离	Wasserstein 距离
$\theta = 0$	0	0	0	0
$\theta \neq 0$	$+\infty$	$\log 2$	1	$\ \theta\ $

从对比中可以看出，Wasserstein 距离有以下两个好处。

- 1. 函数连续。

## 2. 函数的一阶导数不为 0。

这使得它比其他三个度量有更好的表达性质，也更利于优化。

### 10.4.5 Wasserstein GAN 的实现

上面的介绍只是很直观地为大家展示了 Wasserstein GAN 的一些特性。当然这个例子相对简单，Wasserstein 距离的计算也相对简单，对于实际问题来说，距离求解会和它的公式一样异常复杂。为了解决这个问题，Wasserstein 距离需要被转换成另一个相对简单的形式。由于这其中涉及大量超纲的数学知识，本书将不进行详细推导，直接给出结果：

$$W(P_r, P_\theta) = \sup_{\|f\|_L \leq 1} E_{x \sim P_r}[f(x)] - E_{x \sim P_\theta}[f(x)]$$

其中的函数  $f$  需要满足 1-Lipschitz 条件，也就是说，对于定义域内的任意两点  $x$ 、 $y$ ，都有：

$$|f(x) - f(y)| \leq |x - y|$$

对于判别式来说，它的目标是使上面的距离最大化，而上面公式中的  $f$  也可以认为是判别式的形式。在实际训练过程中，由于只需要求出极值， $f$  不需要保持 1-Lipschitz 性质，也可以保持 K-Lipschitz 性质。从公式中可以看出，模型的训练模式发生了变化。在 GAN 模型中，生成模型和判别模型形成了鲜明的对抗，两者的关系势同水火，而到了 Wasserstein GAN 中，判别函数变成函数  $f$  这样的形式，每次训练  $f$  的主要目标是为了让它能够更准确地描述 Wasserstein 距离。由于在上面的公式中，Wasserstein 距离需要取得等号右边公式的上界，因此  $f$  训练的目标就是最大化右边公式。这样看来，曾经的判别模型已经不再具有判别的主要特点，因此在论文中作者也将其改名为评判模型（Critic），于是 10.2.1 节的故事就可以改成下面的版本。

$x$  是一种商品， $G$  是一家山寨公司，希望根据拿到手的一批产品  $x$  研究出生产山寨商品  $x$  的方式。为了保证自己的山寨产品能够通过检验，公司高薪聘请了一个专家团队  $C$ （Critic），这个团队对检验正品拥有丰富的经验，对于  $G$  生产出的产品， $C$  通过不断地研究分析后，能够尽可能准确地告知  $G$ ，他们的工艺和原版的工艺有多大的差距， $G$  则根据  $C$  给出的指导继续改进生产。随着  $G$  的生产质量不断提高， $C$  对两个产品的差异描述也越来越精细，同时对产品的质量要求也越来越高。最终， $G$  生产出足以以假乱真的产品，并通过了  $C$  的评价，专家们终于大声喊出了那三个单词：“I want you!”。这样就完成了训练过程， $G$  产品正式面世。



从上面的描述可以看出，经过 Wasserstein GAN 的推演，曾经的“双人在线对抗博弈”变成了“单人闯关游戏”，因此它形式上也和增强学习有些相近，也确实有研究人员分析过 GAN 和 actor-critic 模型的异同 [9]。

上面的公式经过计算，最终可以得到生成模型参数的梯度：

$$\nabla_{\theta} W(P_r, P_{\theta}) = -E_{z \sim p(z)} [\nabla_{\theta} f(g_{\theta}(z))]$$

如果回过头看看 GAN 模型的第二种目标函数梯度：

$$\Delta \theta = \nabla_{\theta} E_{z \sim p(z)} [-\log D(g_{\theta}(z))]$$

从形式上可以看出两者的相似度还是比较高的，实际上只需要修改判别函数的最终输出和目标函数的形式，我们就可以把 DC-GAN 的模型转变为 WGAN。就是这些差异，造成了效果的不同。从模型的结构上看，Wasserstein GAN 和之前的 DC-GAN 相比有如下差别，如图 10-20 所示。

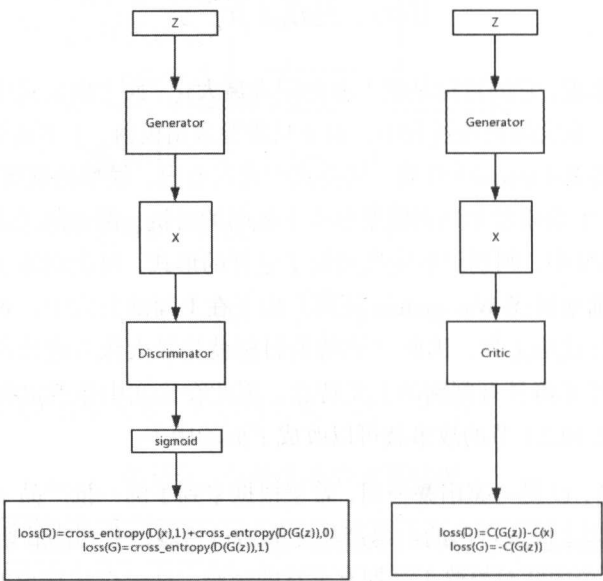


图 10-20 DC-GAN 和 Wasserstein GAN 的模型结构图

从模型结构可以看出，Wasserstein GAN 去掉了最后的 sigmoid 层，在损失函数上有一些区别，除此之外，模型和 DC-GAN 没有明显差别。

最终 Wasserstein GAN 的优化目标由下面两个公式组成。

生成模型：

$$\min_{\theta} -E_{z \sim P_z}[f(G_{\theta}(z))]$$

评判模型：

$$\min_f E_{x \sim P_{\theta}}[f(x)] - E_{x \sim P_r}[f(x)]$$

从实验中发现，Wasserstein GAN 在一定程度上解决了之前 GAN 存在的问题。首先，模型的稳定性变得更好了。由于目标函数的改变，即使真实数据分布和生成数据分布没有 Perfectly Align，生成模型同样可以获得梯度，完成优化，这个现象在理论分析中已经可以看出了。

其次，模型的 Loss 值从过去的相对值变成了现在的绝对值，看上去更能给人建立起一种直观的印象，对于模型的优化更具有指导意义。当然，这里的指导意义主要体现在生成模型的损失函数上。在实验过程中，随着生成模型的生成质量不断提高，它的损失函数值也在不断降低。

在一开始使用 Wasserstein GAN 时，作者为了保持评判模型的 K-Lipchitz 属性，会在每一轮迭代后将参数强制限制在  $[-K, K]$  的范围内，这样会使得模型的优化变得不稳定，动量系的优化算法在这个问题上容易产生不好的效果。后来同一批研究人员又在之前的研究基础上修改了目标函数，新的目标函数变成了：

$$E_{x \sim P_{\theta}}[f(x)] - E_{x \sim P_r}[f(x)] + \lambda[(\|\nabla_{\hat{x}} D(\hat{x})\|_2 - 1)^2]$$

其中的  $\lambda$  用于平衡原始目标函数与新的目标函数的权重， $\hat{x}$  表示了真实数据和生成数据混合而成的数据，混合的公式为：

$$\epsilon \sim U[0, 1], x \sim P_r, \tilde{x} \sim P_{\theta} \hat{x} = \epsilon x + (1 - \epsilon) \tilde{x}$$

通过这样的变换，原本对参数的强制限制变成了对参数更温和的限制，模型的优化也变得更稳定，这样动量系的优化算法又可以重新回到舞台了。更多内容请阅读论文 *Improved Training of Wasserstein GANs*<sup>[8]</sup>。

## 10.5 总结

本章主要介绍了基于深度学习的生成模型，它们在生成图像上有着很强的能力。

- VAE：基于变分下界约束得到的 Encoder-Decoder 模型对。

- GAN：基于对抗的 Generator-Discriminator 模型对。
- InfoGAN：挖掘 GAN 模型隐变量特点的模型。
- Wasserstein GAN：解决 GAN 训练梯度消失、优化不稳定的问题。

## 10.6 参考文献

- [1] Kingma D P, Welling M. Auto-Encoding Variational Bayes[J]. 2014.
- [2] Doersch C. Tutorial on Variational Autoencoders[J]. 2016.
- [3] Goodfellow I J, Pougetabadie J, Mirza M, et al. Generative Adversarial Nets[J]. Advances in Neural Information Processing Systems, 2014, 3:2672-2680.
- [4] Radford A, Metz L, Chintala S. Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks[J]. Computer Science, 2015.
- [5] Chen X, Duan Y, Houthoofd R, et al. InfoGAN: Interpretable Representation Learning by Information Maximizing Generative Adversarial Nets[J]. 2016.
- [6] Arjovsky M, Bottou L. Towards Principled Methods for Training Generative Adversarial Networks[J]. 2017.
- [7] Arjovsky M, Chintala S, Bottou L. Wasserstein GAN[J]. 2017.
- [8] Gulrajani I, Ahmed F, Arjovsky M, et al. Improved Training of Wasserstein GAN[J]. 2017.
- [9] Pfau D, Vinyals O. Connecting Generative Adversarial Networks and Actor-Critic Methods[J]. 2017.

由深度学习引发的新一轮人工智能革命已经在众多领域颠覆了人们的认知,越来越多的人加入研究深度学习的大军。本书详尽介绍了深度学习的基本知识,以及视觉领域部分前沿应用,同时深入分析了工业界十分成熟的开源框架 Caffe,可以帮助读者更快地夯实深度学习基础,跟上深度学习发展的前沿。作者行文在细节上十分认真,书中内容可读性很强,非常适合入门者阅读。

——猿辅导研究总监,邓澍军

近年来,深度学习技术已经给学术界、工业界带来了极大的影响,本书深入浅出地介绍了深度学习基础知识与视觉应用,语言轻松幽默但不失严谨,内容既涵盖经典概念,又包括一些最新的研究成果,特别是对一些底层的具体计算方式有细致的描述,这往往是深度学习入门者忽略的,因此非常适合深度学习的初学者和进阶者阅读学习。

——今日头条 AI Lab 科学家,《推荐系统实践》作者,项亮

随着 GPU、TPU 等专用处理芯片的发展,深度学习技术逐渐从幕后走向台前,开始向世人展现其强大的非线性映射能力。本书从神经网络的基础结构入手,深入分析了深度学习模型内部的算法细节,并总结近年来一些优秀的研究成果,非常适合有志于研究深度学习的初学者和希望快速了解深度学习基础知识与发展的研究人员阅读。

——中国科学院计算技术研究所副研究员,刘淘英

## 作者介绍



冯超,毕业于中国科学院大学,猿辅导研究团队视觉研究负责人,小猿搜题拍照搜题负责人之一。自 2016 年起在知乎开设了自己的专栏——《无痛的机器学习》,发表机器学习与深度学习相关文章,文章以轻松幽默的语言、细致深入的分析为特点,收到了不错的反响,被多家媒体转载。曾多次参与社区技术分享活动。



博文视点Broadview



@博文视点Broadview



责任编辑:郑柳洁  
封面设计:吴海燕

欢迎投稿  
邮箱:zhenglj@phei.com.cn  
微信号:Alinamercy

上架建议:计算机-人工智能

ISBN 978-7-121-31713-2



9 787121 317132 >

定价:79.00元